Tampere University

Henrikki Hoikka

EVALUATION OF ARROWHEAD FRAMEWORK IN CONDITION MONITORING APPLICATION

Faculty of Engineering and Natural Sciences
Master of Science Thesis
December 2019

ABSTRACT

Henrikki Hoikka: Evaluation of Arrowhead Framework in Condition Monitoring Application
Master of Science Thesis
Tampere University
Automation Engineering
December 2019

The technological advancement in the field of electronics and information technology is changing how industrial automation systems are built. This phenomenon is commonly referred to as the fourth industrial revolution. However, before this prophecy on the change can manifest, new architectural solutions are needed to fully leverage the abilities brought by cheaper sensors, more advanced communication technology and more powerful processing units.

The Arrowhead Framework tries to tackle this problem by providing means for Service-oriented architecture via System-of-Systems approach, where so-called application systems consume services provided by so-called core systems, which provide means for service discovery, service registration and service authorization.

The goal of the thesis was to evaluate The Arrowhead Framework by developing a demo application on the edge-cloud setup used in the condition monitoring system of vibrating screens manufactured by Metso. The demo applications objective was to ease the configuration and installation of industrial Linux PC's at the edge of the network.

The methodological model for the evaluation was based on the design science research process (DSRP), which provides a model for research of IT artefacts. As a result, the Arrowhead Framework's core features were found helpful in the problem domain, and suitable for small-scale test setup. However, the implementation of the framework was found to be low quality and lacking features from a production-ready software artefact. The found shortcomings were reported as feedback for the ongoing development process of the framework.

Keywords: Arrowhead Framework, condition monitoring, Service-oriented architecture, REST, Industrial Internet of Things, IIoT, edge computing, containerization

TIIVISTELMÄ

Henrikki Hoikka: Arrowhead sovelluskehyksen arviointi kunnonvalvontasovelluksessa
Diplomityö
Tampereen yliopisto
Automaatiotekniikka
Joulukuu 2019

Elektroniikan ja tietotekniikan kehitys on muuttamassa tapaa, jolla teollisuusautomaatiojärjestelmiä toteutetaan. Tätä ilmiötä kutsutaan yleisesti neljänneksi teolliseksi vallankumoukseksi. Muutoksen mahdollistamiseksi tarvitaan kuitenkin uusia arkkitehtonisia ratkaisuja, jotka kykenevät hyödyntämään halvempia antureita, kehittyneempiä kommunikaatio- ja tietoliikenneratkaisuja ja kasvanutta laskentatehoa.

Arrowhead -sovelluskehys pyrkii ratkaisemaan tämän ongelman palvelukeskeisyyteen ja sisäkkäisiin järjestelmiin (eng. System-of-Systems) perustuvalla lähestymistavalla. Arrowhead -sovelluskehyksessä niin kutsutut sovellusjärjestelmät (eng. application systems) kuluttavat niin kutsuttujen ydinjärjestelmien (eng. core systems) tarjoamia hallintasovelluspalveluita, jotka kykenevät, sovelluspalveluiden rekisteröimiseen, etsintään ja käyttöoikeuksien hallintaan.

Tämän diplomityön tavoitteena oli arvioida Arrowhead -sovelluskehyksen soveltuvuutta Metson reuna-pilvi-mallia hyödyntävässä seulojen kunnonvalvontajärjestelmässä, kehittämällä demosovellus. Sovelluksen tavoitteena oli helpottaa hajautetusti verkon reunalla sijaitsevien, pilveen dataa lähettävien, Linux -teollisuustietokoneiden konfiguraatiota ja käyttöönottoa.

Työn metodologisena pohjana käytettiin suunnittelutieteeseen kehitettyä mallia IT-artefaktien tutkimiseen. Tuloksena, Arrowhead-sovelluskehyksen tarjoamat toiminnallisuudet todettiin käyttökelpoisiksi pienen mittakaavan testiympäristössä, mutta sovelluskehyksen toteutus todettiin kuitenkin heikkolaatuiseksi ja sen toiminnoissa havaittiin puutteita, jotka tekevät toteutuksesta käyttökelvottoman tuotantoympäristössä. Löydetyt puuteet kirjattiin ja annettiin sovelluskehyksen edelleen jatkuvan kehitystyön käyttöön.

Avainsanat: Arrowhead-sovelluskehys, kunnonvalvonta, Palvelukeskeinen arkkitehtuuri, REST, teollinen IoT, Säiliöinti

PREFACE

I want to thank Assistant Professor (tenure track) David Hästbacka for the help he gave while I was seeking a topic for the thesis, and for the advice, he gave in the roles of the supervisor and the examiner during the writing process. Thanks also to the second examiner, Professor Matti Vilkko, who gave valuable advice in the initial meeting held at the university.

From Metso's side, thanks to M.Sc Tech. Antti Jaatinen, who in the role of a supervisor helped along the process. Thanks also to M.Sc Tech. Janne Kytökari and PhD. Cristiano di Flora whom both found time for answering my questions.

Finally, thanks to my family for the support they gave during my studies.

In Tampere, 1st December 2019

Henrikki Hoikka

CONTENTS

1 INTRODUCTION

In a world where computing power is cheaper every year, the urge to connect devices in more smart ways is growing. In everyday life, involving home electronics, cars and whatnot, this phenomenon is usually referred with the term Internet of Things (IoT). However, this phenomenon is not limited to smart refrigerators and cool gadgets like the Amazon Alexa. The way how industrial systems and societally critical infrastructure such as the electric grid are implemented is also under change.

For the last 20 to 30 years, the architecture of software systems in the field of automation have been based on the hierarchical model. In this model, automation systems are constructed in a pyramid-like structure, where sensors, actuators and controllers live on the ground level, and more abstract functionalities, such as business planning and logistics live at the highest peak of the pyramid. While this hierarchical approach gives a solid base for software development, it also introduces caveats by tightly coupling different parts of the system to their respective place in the hierarchical model. [13]

On the field-level in industrial systems, the change caused by technological advancement propagates as more sensors and therefore way more data-points. Measurement as such is not the point though. The data produced by different kinds of sensors need to be processed, moved and acted upon accordingly. The increase in computing power and the advance in communication technologies means that there are more and more choices where to act, process, and where to move the data. For example, some data-processing task that previously would have been done in some computer more central to the system as a whole might be done on the sensor producing the data, which after the processing, sends more refined and abstract data forward.

The change puts the model in which we build industrial systems, under a severe need of rethinking. In a computer system where bits and pieces are tightly coupled to a hierarchical model, it is hard to utilize new communication and data processing opportunities. More dynamic and loosely coupled ways to build more scalable industrial systems and ways to add, move and replace computing units both vertically and horizontally are needed. I.e if it otherwise makes sense[1] to do a computational task closer to a sensor, in a cloud somewhere else, or a parallel unit on the same level, the architectural model should not prevent this from happening.

[1] Sense in a context of the system and good engineering practices in general, such as taking into account things like security, safety, reliability and usability.

In the field of automation, these new requirements haven't been unnoticed. There are multiple different organizations and working groups and projects with participants from the public sector, private sector and the academia, trying to tackle this very problem described above. One major project is the Industry4.0 working group initiated by the German government [62].

1.1 Introduction to Case Metso

This thesis is part of EU funded Productive4.0 project, of which Metso Oyj, the commissioner of the thesis, is a participant of. Like many other projects, Productive4.0 has partners and stakeholders from universities, companies and state-owned research facilities from all over Europe.

Productive4.0 is a continuation for a previous project called Arrowhead, in which stakeholders from all over Europe were collaborating to define a framework that eases the utilization of service-oriented architecture in the field of automation and embedded computing systems in general.

The end product of the project, the Arrowhead Framework, is tackling the problems caused by the changing field, with System of Systems approach, in which so called Arrowhead core systems provide the means for service discovery, service registration and service authorization, for so called Arrowhead application systems, which together form the aforementioned System of Systems. Communication between the systems is handled via REST-based[22] web services.

In Productive4.0, the participants examine ways to do Service-oriented architecture (SOA) in a spirit of Industrial internet of things (IIoT). On Metso's side, the project chosen to participate in Productive4.0 is condition monitoring system used in equipment manufactured by Metso. The "test-bench product-line" for the project is vibrating screens, used in mines, but the possible reusable results in the context of other product-lines by Metso found during the project, are welcome, and to a some extent even expected.

1.2 Problem Definition

The condition monitoring system which is the objective in one of the tasks of the Productive4.0 project is already functioning, and it utilizes modern techniques such as Bluetooth LE, energy harvesting and distributed analysis of the measurement data collected from accelerometers. However, the caveats caused by hierarchical architecture mentioned in the first part of this chapter are at least partly present in the system.

Configuring the systems data flow is found cumbersome, and control over what part of the analysis is done at what level of the process, is found to be tricky to configure. Ideally, the mutability of what is computed where, and how the data flows, would be far higher than it is in the current system, which would enable ways to change how the system is functioning dynamically, ability to expand it, and therefore increase Metso's ability to use data to

achieve more responsive maintenance, and give feedback to product development.

Another major drawback in the current system is the installation of new edge devices — which are industrial computers on the premises, running Linux. When a new edge computer is installed, or a broken one is replaced, the installation and initial configuration of the condition monitoring system is found to be tricky. Since condition monitoring is an extra service on top of the actual purpose of the machine, the extra work needed to keep the system functional in all situations should be minimal. In an ideal world, the addition or replacement of an edge computer should happen in plug and play nature.

1.3 Research Questions

This Thesis has two research questions which are both derived from the problem definition stated in the previous section:

- How can Arrowhead Framework help in configuration of the data flow from edge to cloud?
- How can Arrowhead Framework ease the installation of new edge devices?

To answer these research questions, the Arrowhead Framework is used to develop a demo application that tries to tackle the problems. After the implementation, Metso's research facilities located at Tampere are used to examine the demo application with a vibrating screen exciter and edge computing devices.

1.4 Structure of the Thesis

In chapter 2 related research and theory in the domain of the thesis is summarized in the form of a literature review. In chapter 3 tools used for the demo application are presented. In chapter 4, the methodology and facilities used for solving the research questions are introduced in more detail. Chapter 5 is about implementation of the demo application. In chapter 6 implementation is evaluated, and as a result, research questions are answered. Chapter 7 concludes the thesis.

2 BACKGROUND

In this chapter, a literature review on the domain of the thesis is concluded. In the two first sections, some theory behind maintenance approaches and condition monitoring is presented. However, while understanding some basic concepts of maintenance approaches and condition monitoring is crucial for understanding the domain of the thesis, they are not the main point of interest. There are multiple good sources for information on deeper level [2][15][56].

The second section is about the Internet of Things(IoT), and some background on IoT, industrial IoT, IoT architecture and concepts relevant to the thesis is introduced. The third section continues where the second one ended and goes into more depth on the domain of the thesis. Theoretical justification on the partly pre-chosen approaches used are presented in the form of improvements to the approaches presented in the second section.

2.1 Maintenance Strategies

The main goal of maintenance is to restore the equipment to a state, in which it can fulfil its designated task. Maintenance is an essential part of the modern production line, and proper maintenance strategy can potentially reduce failures, increase runtime and therefore decrease costs [2, 56].

Maintenance approaches can be loosely categorized in two different classes, Corrective Maintenance(CM) and Predictive Maintenance(PM) [15, 56]. In CM, the goal is to repair the equipment after a failure has already happened. Whereas In Predictive Maintenance, the goal is to estimate or monitor the condition of the equipment and repair it before it breaks [3, 56].

CM can lead to production loss due to unexpected downtime caused by a failure[56]. For minimizing the number of unplanned stoppages, different strategies to predict when the maintenance is needed have been developed. The greatest challenge in these PM based strategies comes from the difficulty of predicting the optimal time for maintenance. To do so, lots of data, and knowledge "mined" form the data is needed [2].

The prediction process can be based on experience gained while using the equipment or scientific method which can be based on on-site-analysis or recommendations given by the original manufacturer of the equipment. In PM with the experience-based method,

things are not necessarily done in a systematic nor standardized way. The staff and its experience gained while using and repairing the equipment over time is an important role. However, since no human being can work around the clock, and staff changes over time, this can lead to situations where one or multiple members of staff become crucial to the process. While the most experienced members of staff are not available, proper actions can not be taken [2].

2.1.1 PM Based on Scientific Method

The methods based on a scientific approach can be further divided into two separate categories, time-based maintenance (TBM) and Condition Based Maintenance (CBM). In TBM the failure data of the equipment is analysed, and based on characteristic found in the data, recommendations on future maintenance schedule are given. In CBM, the data representing the state of the equipment is collected and analysed on-fly, and maintenance recommendations are given based on the monitored state of the equipment. The gathering of the data and on-fly analysis can be performed by staff with special tools engineered for the job or by equipment that is by design, or by addition, an integral part of the system [2].

In the TBM method, the main flaw is in its incapability to be general enough solution in all cases. For example, in the case where the original manufacturer gives the recommendations, the difference in the environment where the equipment is used can lead to high variance between the time when maintenance is genuinely needed [2].

In the worst-case scenario, the difference between maintenance actions taken is too short, and the equipment will break before the maintenance is scheduled. On the other hand, the schedule might also be too conservative, which leads to too early maintenance, in situations where the state of the equipment was still acceptable for a longer run. [2]

The primary source of problems in TBM is the assumption that the failure of a system can be represented with age-based models like the bathtub model presented in figure 2.1 [2]. Bathtub model assumes that trends in failure rate can be divided into three periods:

- Burn-in or infant mortality period
- Useful life period
- Wear-out period

Several independent studies show that age-based maintenance strategies are suitable for only 15 to 20 per cent of cases [3]. In other words, the majority of the failure-rate profiles are not as age-dependent as the bathtub model assumes. Therefore, [3] concludes that CBM based methods should be used instead, in cases where the bathtub does not describe the failure rate of the equipment.

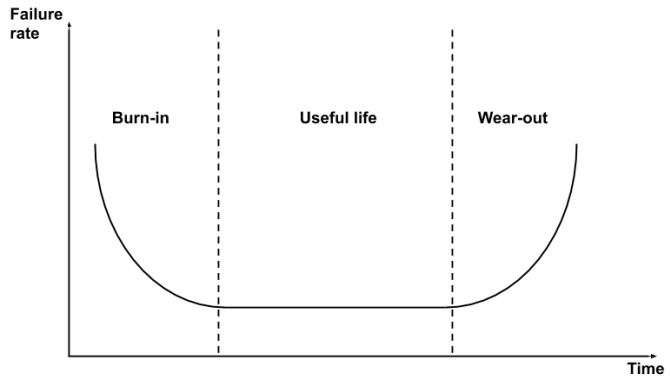

Figure 2.1. *In TBM it is assumed that bathtub curve represents a failure rate of ageing equipment*

2.2 Condition Based Maintenance

Since 99 per cent of mechanical failures can be predicted from indicators before the failure happens [7], it is justifiable to monitor these indicators and make the maintenance decisions based on the state of the equipment. If the monitoring and decision-making are done successfully, physical abnormalities can be detected, and maintenance actions could be taken only when they are needed [29].

Vibration monitoring is the most commonly used monitoring technique in industry, especially for rotating machinery. In vibration monitoring, the vibration data is collected from the equipment either with a handheld device designed for the task or by instrumenting the equipment with sensors that can collect the data automatically. Vibration monitoring is based on the assumption that wear of the equipment can be detected from changes in frequency spectrum [2].

Other commonly used techniques are lubricant monitoring and acoustic monitoring, where the former tries to detect possible wear from contamination of the lubricant and the later tries to find marks of incoming failure from acoustic noise[2].

2.2.1 Phases of CBM

The CBM workflow can be divided into three steps; data-acquisition, data-processing and decision-making [29]:

Data Acquisition

In the data acquisition phase, the data is collected from the monitored equipment. The data can be separated into two categories, event data and condition monitoring data [29].

The event data usually requires manual entry and represents the events that have hap-

pened to the equipment, for example, maintenance actions that were performed [29].

Jardine et al. argue [29], that while the event data is mistakenly considered as less important compared to condition monitoring data, both are essential. Event data can be seen as feedback for the effectiveness of the current condition indicators and decision-making. If the condition monitoring system is able to monitor effectively, the events should be in line with the monitored condition, and the number of surprises in event data should be minimal. However, since manual data entry is needed, the erroneous event data is still unavoidable due to human error, and this should also be taken into account.

Data Processing

The data processing phase begins with data clean up. The clean up can be a tedious task and generic enough algorithms to automate the process are hard to develop, and data clean-up may need manual intervention[29].

After data is cleaned, the next step in the data processing phase is to perform analysis on the data. The point of analysis-phase is to extract interesting features from the data. The way the analysis is performed depends on what kind of data was collected in the data acquisition phase on the use case at hand [29].

In case of vibration analysis, the analysis methods are usually separated into three domains; time, frequency and combined time-frequency domain [29]

In the time domain the waveform itself is analysed and descriptive statistics, like peak-to-peak values are extracted from it. One popular way is time-synchronous average; it tries to reduce noise by producing an average of the signal on the span of multiple sample intervals [29].

In frequency domain analysis, the signal is transformed from the time domain to the frequency domain most commonly with Fast-Fourier Transform (FFT). After this, the signal can be analysed on the level of different frequency components. One typical example is the envelope analysis, which can be used to extract periodic impacts caused, for example by deformation on the race of a bearing [9, 29].

In time-frequency analysis, both domains are used. It allows non-stationary phenomena, common in failing equipment, to be detected with more ease, compared to sole frequency analysis. One way to do this is to perform wavelet transform and analyse the signal with various methods developed for wavelets [29].

Decision making

Decision making in condition monitoring can be divided into diagnosis and prognosis. In diagnosis, the goal is to find early signs of wear from the equipment, whereas prognosis aims to estimate when the equipment will fail [2, 29].

Diagnosis is pattern recognition, where the goal is to map features extracted in the data

processing phase to fault types. It can be done both manually or with tools that automate the process. However, since manual pattern recognition requires lots of work and skill, automating the process is preferable [29].

Some commonly used methods for diagnosis include statistical approaches like cluster analysis, approaches based on artificial intelligence like neural networks and approaches based on mathematical modelling [29].

In prognosis, the most common approach is the prediction of the time when failure will occur, based on the current state and operation profile of the past. This is also known as the remaining useful life (RUL) [27, 29].

2.3 IoT

The Internet of Things refers to a vision, in which interconnected physical everyday objects, "the things," can sense their state and their environment. The primary enabler for the vision is the advancement in information and communication technology (ICT), which has made the price of computing lower, year by year [36].

Internet of Things has similarities with the concept of ubiquitous computing (UC), introduced by Mark Weiser in the early ’90s at Xerox PARC Labs. In UC, objects in the real world have embedded computational elements and capability to communicate over network [58].

Today, almost 30 years from the introduction of Weiser’s vision, the ongoing progress has made the vision achievable, and while the realization of IoT concepts like smart cities and other similar large scale ideas are still in the future, the first steps in the form of various "smart" gadgets have already been taken [36].

The vision of IoT is broad, and the description of what it contains and what could be possible due to it varies. The earliest descriptions see IoT mainly as the usage of technologies like Radio Frequency Identification RFID [36], which enables identification of objects in distance of few meters. The more recent definitions have a broader context and include more technologies, for example, Wireless Sensor Networks (WSN) and cloud computing which are briefly covered later [36]. However, it is widely accepted that the realization of the vision will have a significant impact on our lives [36][31][62][61].

Since IoT is growing fast, the standardization of the technologies involved is hard. Specifically, issues in radio access, security, interoperability and privacy are areas where standardization is needed. Successfully tackling the issue of standardization would enable products from different vendors to be used together with more ease, and therefore, further accelerate the realization of the vision [61].

2.3.1 Wireless Sensor Networks

The concept of WSN refers to things, with embedded sensing ability, connected through various network technologies. Individual things in the network are commonly referred to as nodes, which have the capability to process data on their own, enabling the distribution of computation [25].

Since communication stacks between different WSN subnets vary, gateways, which enable integration between subnets are used [25]. Use cases for WSN's include, for example, industrial, traffic and environmental monitoring [61].

2.3.2 Cloud Computing

According to American standardization authority, National Institute of Standards and Technology(NIST), cloud computing is defined as follows [37]:

"Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction."

For a while, enterprises have been moving away from IT-solutions based on traditional ways of renting virtual servers. Pay-as-you-go model, in which the billing is based on usage of resources, offered by cloud vendors, has allowed users to elastically change the amount of computing and storage resources without extra costs for maintenance [23].

Cloud solutions are sold as resource-based services, with service models that differ on level how close to a ready application the service is. The three service models are [23]:

- Infrastructure-as-a-Service (IaaS) — virtual machines, network gear, such as virtual routers, and other infrastructure level resources.
- Platform-as-a-Service (PaaS) — ready to use, automatically scaling platforms on which user can deploy their applications, without the need to take care of the infrastructure level resources.
- Software-as-a-Service (SaaS) — ready to use applications.

On top of these basic models, the users can also offer their own products as a service. For example, In an IoT use case, Sensing-as-a-Service could be offered by integrating WSN with the cloud and offer its data through API and user-interfaces. In condition monitoring use case Monitoring-as-a-Service could be offered. In this service, the manufacturer of the monitored equipment could offer prognosis, diagnosis and easy access to other refined analytical data related to the condition of the equipment [23].

2.3.3 Industrial IoT

In manufacturing and field of automation emerging of IoT and the similar concept of Cyber-Physical Systems (CPS) is going to lead in massive change [59]. The change is referred to as the 4th industrial revolution. The three previous revolutions are commonly presented as follows[62]:

- 1st, mechanical systems, powered by steam and water, the early 1800s
- 2nd, mass production, powered by electricity, the late 1800s
- 3rd, automation, enabled by electronics and Information technology, the mid 1900s

Many countries have established initiatives to prepare for the incoming change. Most commonly known, is the German Industry4.0 Initiative, established in 2011. The Industry4.0 aims to achieve smart manufacturing, and it is expected to transform industrial ecosystems in a wide range of applications [63].

One use case that the change is enabling is more accurate condition based maintenance, which was briefly introduced in section 2.2. Some examples of research on IIoT and condition-based maintenance include; a fog computing-based monitoring framework by Wu et al.[60], a monitoring solution for predictive maintenance by Civerchia et al. [10] and a paper by Halme et al. in which the Arrowhead Framework, the framework evaluated in this thesis, was used as a part in a conceptual model for condition monitoring [26].

The Enterprise Resource Planning (ERP) and Manufacturing Execution Systems (MES) are also moving away from the traditional pyramid-model of the automation. Local manufacturing clouds, which mostly are on-premise data-centres, are used more often today. Value chain may also include private inter-enterprise cloud solutions, which allow the direction of data-flows to various stake holders [1] and their private enterprise clouds. Compared to the previous solutions, this, for example, enables more visibility to the customer on the movement of products while they go through the factory floor [59].

While there is a change in the state-of-the-art, in Industrial automation, change is slow. This is mainly due to the price of the systems, preference on reliability and the mentality of "not fixing it if it is not broken". The life cycle of automation systems might be as long as 40 years, and systems added later, need to co-exist with the older systems, which are using technologies like various field-buses and Industrial Ethernet implementations for communication. The slow phase makes the issue harder on Industrial domain compared to more general "every day" IoT. Also, other requirements mainly on safety, security and reliability are much higher [59].

2.3.4 Architecture of IoT

Usually, the architecture of IoT is presented or visualized with a layered model. Most common is the three-layered model presented in figure 2.2 [31, 61] and the different

[1] for example customers and equipment providers

architecture layers are described below:

- Perception layer — is responsible for interaction on the physical level. It is the layer closest to Sensors, actuators and RFID-tags. Due to increased processing power, different nodes on the perception layer commonly possess the ability to process the data they are collecting or acting upon in case of sensors and actuators respectfully [31].
- Network layer — is responsible for transmitting the information, including Integration of various communication technologies and hardware, like gateways, which enable the communication between, multiple types of networks and form one "web", in which things and applications using different technologies can find each other [31].
- Application layer — is responsible for the "business logic" and interfacing with the user. One example of application is a logger, which logs the data collected from sensors, stores it to database, and provides access to historical data for the user. If needed, more features could be added, for example, a user interface for visualization [31].

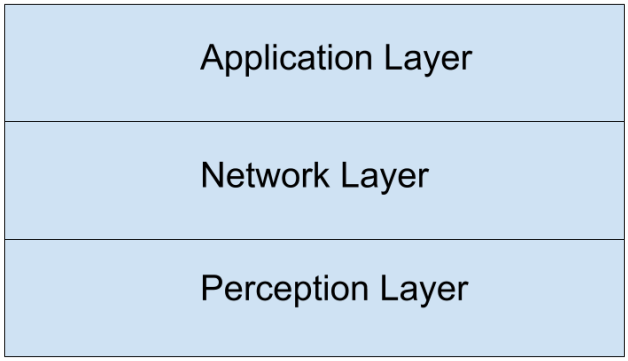

Figure 2.2. *Layered architecture of IoT [31]*

2.4 IoT in the Domain of the Thesis

The three-layer architecture presented in figure 2.2 does not come without problems. The roles of different layers are quite broad and vague, especially in the case of application and network layers. Also, the integration to the cloud, which has services as its primary way to abstract resources, might be cumbersome [31].

While cloud computing has its upsides, for example, the way of handling services as everyday commodity objects, or the seemingly infinite and elastically auto-scaling com-

putational resources, in industrial use-case, some things just do not fit in the model of handling everything elsewhere, possibly thousands of kilometres away [52].

In cases where lots of data is transferred. The remoteness of the cloud will cause a bottleneck, and limit the functionality of the system as a whole. The problem gets even worse, in cases where the connection to the cloud is bad. Authorization and security can also become issues.

This means that in some cases local computational resources are still needed, which on the other hand brings new problems on how the services running in cloud and on-premises should be deployed, integrated and composed [31]? The problem is the same as with the architecture of IoT: How to integrate various distributed computing resources and the cloud — which undoubtedly has its upsides?

To mitigate the problems described above the following solutions are suggested:

- Additional service layer to architecture presented in figure 2.2 to enable Service-oriented Architecture, which simplifies the roles of other layers.
- Edge computing as an extension for a cloud to allow computation and data storage on-premises, which allows more control on data refinement, data caching, and data access.
- Containerization for the deployment of services, to both edge and cloud, to allow more clear and finely grained control-interface on what services are deployed and where.

Solutions enlisted above are introduced briefly in the following sections.

2.4.1 Service-oriented Architecture

Service-oriented architecture refers to an approach to design computing systems with services as its main abstraction. Services are interoperable, reusable and loosely coupled artefacts, that can be used to compose larger systems [19, Chapter 4].

A wider umbrella term, service-oriented computing, defines a set of goals, of which most relevant in the context of engineering are; increased Interoperability, increased federation and increased possibilities to choose the vendor. To achieve these goals service orientation is applied to a problem. In service orientation, eight design principles are defined [2] [19, Chapter 4]:

- Standardized Service Contract — "Services within the same service inventory are in compliance with the same contract design standards."
- Service Loose Coupling — "Service contracts impose low consumer coupling requirements and are themselves decoupled from their surrounding environment."

[2] All are direct quotations from [19, Chapter 4].

- Service Abstraction — "Service contracts only contain essential information and information about services is limited to what is published in service contracts."
- Service Reusability — "Services contain and express agnostic logic and can be positioned as reusable enterprise resources."
- Service Autonomy — "Services exercise a high level of control over their underlying runtime execution environment."
- Service Statelessness — "Services minimize resource consumption by deferring the management of state information when necessary."
- Service Discoverability — "Services are supplemented with communicative meta data by which they can be effectively discovered and interpreted."
- Service Composability — "Services are effective composition participants, regardless of the size and complexity of the composition."

As an output of applying the design principles enlisted above, one gets "service-oriented" services, which can be further used to form an SOA. It is also important to note that services created by using service orientation are not limited to any specific technology, even on one system. For example, the services offered in different service models of cloud computing, introduced in IoT section 2.3, can be a result of the same service orientation process [19, Chapter 4].

Service Oriented IoT

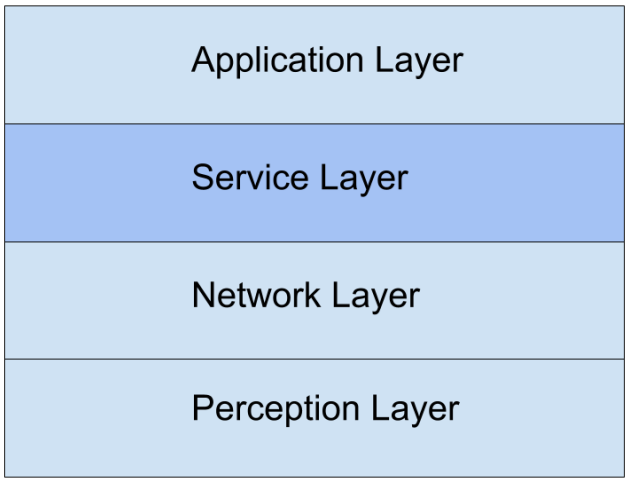

Figure 2.3. *Updated Service-oriented architecture for IoT [31].*

After applying service orientation to the IoT architecture presented in figure 2.2, the new architectural stack looks as is presented in figure 2.3. The new service layer is responsible for service discovery, service composition, service management and interfacing

between services. On this layer, matchmaking between service providers and service consumers is taken care of, while making sure, that requirements on quality and security are met [31].

The layer allows a more finely grained approach on integrating the physical world with the cloud. Application-level and network-level are also relieved from the various service like features. For example, in the case of the application layer, in the previous model, the layer needed to handle all the integration between the devices living on the perception layer, including the users and devices needing the data. In the new model, the application layer can be used to interface with users and data sources, while the service layer takes care of the "how" [31].

It is also important to note, that the number of layers and their exact roles vary. In some papers it has been presented that the amount of layers should be as high as 7, where some layers are more granular versions of the model presented above and some take new roles similarly like the service layer takes compared to the older three layered model. Burhan et al. analysed various layered models of the IoT [8].

REST

Roy Fielding defined representational state transfer (REST) in his doctoral dissertation [22]. It is an architectural style, in which representation of the state, commonly referred to as a resource, is the main abstraction. Although REST and SOA are both architectural styles, they are not in clash with each other on the level of principle. Therefore, SOA can be implemented with REST as its medium [19, Chapter 7].

Fielding defines five constraints and one optional constraint on REST [22]:

- Client-Server — Clients are separated from servers, which responds to requests made by clients.
- Stateless — Request must contain all the information needed to full fill it. The server does not store client state.
- Cache — Caching of responses, must be definable either implicitly or explicitly in requests, in case of equivalent requests in the future.
- Uniform Interface — REST sets four constraints on interfacing: identification of resources, manipulation of resources through representations, self-descriptive messages and hypertext as the engine of application state (HATEOAS).
- Layered System — Hierarchical layers, which are visible only to the next layer.
- Code-On-Demand(optional) — Scripts can be sent in responses, for clients usage. Mainly for web-browsers.

From these constraints, the Uniform Interface is the vaguest and needs more specifying. The "identification of resources" refers to the mapping of resources to Universal Resource Identifiers (URI). The "manipulation of resources through presentations" refers to meta-

data which can be used to manipulate the resource, for example, metadata that specifies is the resource preferably in JSON or XML form. "Self-descriptive messages" is related to the stateless constraint. The "hypermedia as the engine of application state" refers to a paradigm where hypermedia links are sent within the representation of resources, which allows the client to use them in state-machine-like fashion. The most common example of this is the HTML browser, which is used by following the links, embedded in HTML document sent by the server as a response [22].

The most common way of implementing REST is via mapping resources to a Universal Resource Identifier (URI) which is manipulated via standard HTTP methods (for example GET, POST, PUT and DELETE) and headers. Another protocol that is compatible with REST is Constrained Application Protocol (CoAP) [51], it is mainly designed small scale devices in mind, while still making it integrable with the HTTP, which is the main "language" of the web [31].

Web-services implemented with REST are sometimes called RESTful-services. It is also worthwhile to note that the terms REST or RESTful are not always in practice used when referring to services that satisfy all the constraints listed above.

REST is not the only way of implementing web-services. Another commonly used approach is the Simple Object Access Protocol (SOAP). The way how a certain SOAP service is used is commonly described with documents in Web Services Description Language(WSDL) [24]. Similar schemes have also been developed to describe REST-based services; one example of this is the OpenAPI [43]. However, in the REST world, the declarative description of the services is not as a fundamental part of the "service scheme" like it is often with the SOAP-based ones [24].

2.4.2 Edge and Cloud

Edge computing refers to an approach, where computational tasks and data storage is moved to the edge of the network, towards the producer of the data. The main motivator behind this is the reduction of the amount of data, that needs to be transported over the internet, which mitigates the bottleneck that is caused by latency and low throughput of the internet [52].

The location of the edge is presented in figure 2.4. In the picture, data producers, for example, sensor nodes of WSN, are connected to a gateway, which has a passage to other gateways and the cloud. The various devices, with computational capabilities in the edge, are sometimes called edge devices or edge computers, which could include anything that has processing power and ability to connect to other devices.

Other similar concepts to edge computing include fog-computing, mist-computing, core-computing, local-clouds and cloudlets. However, while having subtle differences, mainly in how far from the edge the computing is performed, all share the same goal of moving computation away from the cloud [31, 52].

The benefits of edge computing are not limited to data transfer. Since the computation is performed closer to the edge, the saved bandwidth could be used to connect systems of stakeholders straight to the edge and provide data views, according to roles and needs of the particular stakeholder. The views could be used to form a collaborative edge, where various stake holders offer views to their part of the edge, and therefore enable smoother collaboration between domains [52].

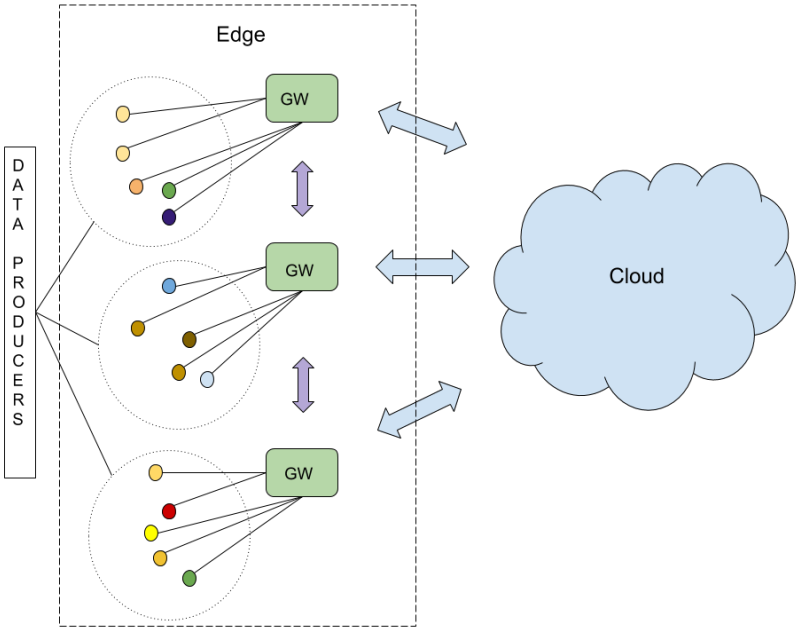

Figure 2.4. *Location of the edge.*

2.4.3 Containerization

Containerization is a virtualization technique which allows light weighted and easily movable environments that capture their dependencies and configuration. Containerization allows easier project management, and deployment via separation and tooling build for management of the containers [17]. When operating system concepts are mentioned in this review, Linux is assumed.

The most common virtualization techniques are presented in figure 2.5. The traditional way of virtualization requires so-called hypervisor, which is responsible for creating a virtual environment, on which the virtual guest systems can run on. Hypervisors are separated into two types; "type 1" and "type 2". In type 1, the hypervisor runs directly on the hardware, while in type 2, the virtualization is done on top of the host operating system, where the hypervisor is run like any other program. Modern CPUs have instructions for virtualized environments to gain full access to kernel-mode instructions from the user-mode, which are leveraged by the hypervisor running on top of the host operating system, to achieve better performance by avoiding the need to trap to the host kernel [17].

The third way in figure 2.5 is the container-based virtualization. Unlike the hypervisor, the

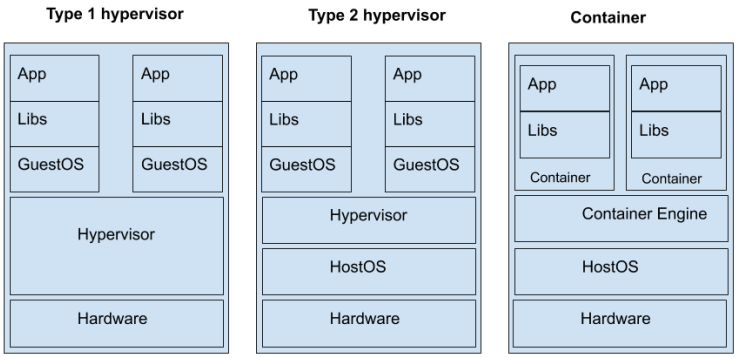

Figure 2.5. *Different virtualization solutions*

containers run directly on top of the host operating system and achieve the isolation by leveraging kernel features, mainly namespaces and control groups. To ease the access to features, which enable containers, various tools like Docker [14] and Linux Containers have been developed [17].

Namespaces provides means for separating the resources of a process from global resources. Linux has namespaces for mount-points, process IDs, control groups, network utilities, shared memory regions, usernames and hostnames. In practice, this means, that the kernel provides the capability of having multiple separated namespaces for each of the resources, which makes the resource seemingly unique to the process that is mapped to them [34].

Control groups offer means for restricting resource usage of processes. The user can, for example, restrict the usage of CPU, memory, disk or network I/O. Control groups are compatible with namespaces [34], and therefore, resource usage per namespace can be restricted. Apart from restricting the usage of resources, control groups are also capable of monitoring the usage of resources, and "snapshotting" the state of a process to freeze and restart them promptly [11].

Apart from providing tooling that offers control over the containerization functionality itself, also more abstract ways of using the containerization ability of the kernel exists. For example, so-called execution engines that enable serverless paradigm, where the main unit of abstraction is a function, are commonly built by using containerization behind the scenes [45][1].

3 TOOLS

In this chapter, tools and software used in the demo application are introduced. In the first section, the primary tool — the Arrowhead Framework — and philosophy behind it, is presented. While evaluation of the Arrowhead Framework in condition monitoring application is the primary goal of this thesis, the framework alone is not providing a full set of tools needed for the demo application. Because of this, a set of tools that enable the evaluation are presented at the end of the chapter.

3.1 Arrowhead Framework

The Arrowhead Framework aims to provide means for integrating, developing and deploying interconnected systems in the field of automation and embedded systems in general. It was originally developed in the Arrowhead Project, which was a large collaborative project funded by the European Union. The project had 80 partners and a budget of 68 million Euros [57][5].

The development of the framework continues in Productive4.0, which, like its predecessor, is also EU funded. The Productive4.0 is a large project with 109 partners from 19 different countries with a total budget of 106 million Euros.[48]

In Productive4.0 the work is distributed in ten working packages. The Arrowhead Frameworks development and research work is done at the working package 1 of the Productive 4.0 project [48]. Since Arrowhead Framework is still a work in process, the target in this brief review is the version 4.2.1, which was the most recent one while this thesis was started[4].

3.1.1 Philosophy of Arrowhead Framework

Arrowhead Frameworks design is based on a service-oriented system of systems philosophy. To put it more clearly, while Arrowhead Framework is utilized the business logic of software is implemented as systems, which provide and consume services from each other. The systems consuming each other's services form a system of systems by utilizing the services provided by the so-called core systems, which provide means for service registration, service discovery and authorization.

Currently, REST is the default style for services in Arrowhead Framework, and both se-

cure HTTPS and insecure HTTP are supported. However, there is an urge also to support other protocols and styles of services, mainly via wrappers. To name few; XMPP, COAP and OPC-UA. One additional goal is to encourage the wrapping of legacy systems [57].

The systems implementing the services are often separate executables or in case of a platform without operating system, the firmware. The system of systems can be seen as distributed computing where the group of individual systems can run on the same or different physical or virtual computing platform, ranging from small scale embedded devices to high-end servers [57].

The concept of local cloud is presented in figure 3.1. In the Arrowhead Frameworks context, the term local cloud is used to refer to the bundle of application systems under the control of the same core systems, which are introduced later [57]. The concept has overlapping with the concept of edge computing, in cases where the Arrowhead local clouds are at the edge of the network, which can be seen as its primary domain. However, despite their name, the local cloud instances can also be run in the more traditional cloud, which enables the introduction of edge cloud architecture [57].

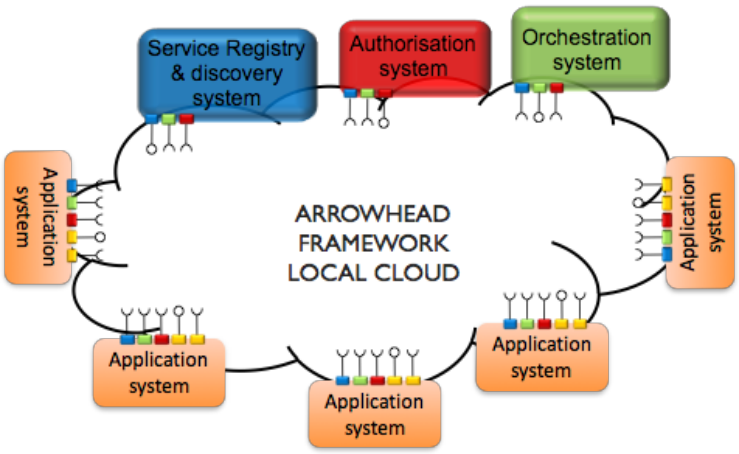

Figure 3.1. *Local Cloud concept of Arrowhead Framework [5].*

The core systems are mandatory for a local cloud and can be seen as the manifestation of the framework. The core systems try to answer the following questions [5]:

- How a service provider can announce its existence for potential consumers?
- How a service consumer can discover available services?
- How the service provider and service consumer decide which is a suitable provider or consumer for them?
- Who is authorized to consume services offered by whom?

Service Registry

The service registry is responsible for keeping track of what service is provided by which application system. When a service provider starts its execution, it should register its

services to the service registry. Vice versa, once the service provider stops its execution or service otherwise comes unavailable, the system should deregister its services [5].

While issuing the registration, the registering system can define certain restrictive parameters on its entry document. These include things like document types that are supported, arbitrary metadata key-value pairs and timestamp on which the registration should be considered as expired [4].

In the paper by Varga et al. [57], where the core systems' architectural design was introduced, the goal was to implement service registry by utilizing DNS-SD, but in the latest version of the framework, the implementation of the service registry is done by storing the service entries in a MySQL database [4].

Orchestrator

The orchestrator core system is the most central system in the Arrowhead Framework. Through the orchestrator, application systems discover each other's services. During service discovery, the orchestrator system consults the service registry on behalf of the consumer and after a provider is found, before responding the result to the consumer, it makes sure that the consumer system is allowed to consume the service by consulting the authorization system, introduced in the next section [4].

Similarly to the service registry, the orchestrator system can reduce the set of potential providers via restrictive parameters set by a consumer in its service discovery request. These include things like, supported document types, metadata key-value pairs and name of the preferred provider [4].

In case gateway and gatekeeper systems, which are introduced later, are in use. The orchestrator can leverage these systems and issue service requests to the neighbouring local clouds and allow so-called "intercloud service discovery". Consumers can also prevent this behaviour by explicitly forbidding it on their service request, in so-called "orchestration-flags" which offer some control on the orchestration process [4].

Authorization

The authorization process is presented in figure 3.2. The authorization control step makes sure that the consuming application system is authorized, by checking whether an entry for that particular consumer-provider-pair on the requested service exists in the MySQL database. The user has to explicitly add a row in the database for each application system pair that should be authorized [4].

In the secure HTTPS mode, after a successful authorization control step, the token needed for communication between the provider and the consumer is generated. In the insecure HTTP mode, the communication does not require tokens, so the generation step is skipped [4].

Besides being responsible for authorization between systems on the same local cloud, the authorization system also controls the authorization of service discovery requests coming from foreign local clouds, through the gatekeeper system, which is introduced later. In this case, the authorization control step makes sure that the local cloud where the request is coming from is authorized, by checking whether an entry for that particular local cloud-provider-pair on the requested service exists in the MySQL database. If this entry exists, every willing consumer in the foreign cloud is authorized to consume [4].

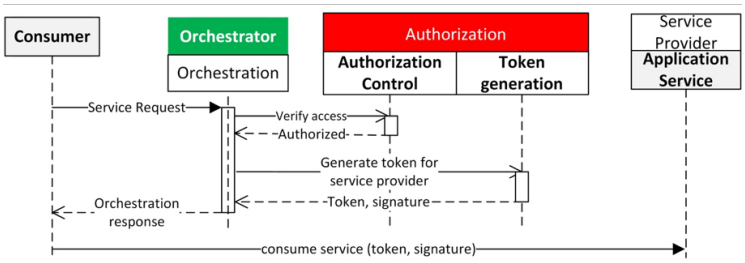

Figure 3.2. *The authorization process performed during the service discovery [4].*

3.1.2 Application Systems

In the terminology of the Arrowhead Framework, application systems correspond to systems developed by the user. These systems implement the business logic of the Arrowhead system of systems which is formed with the help of the core systems [5].

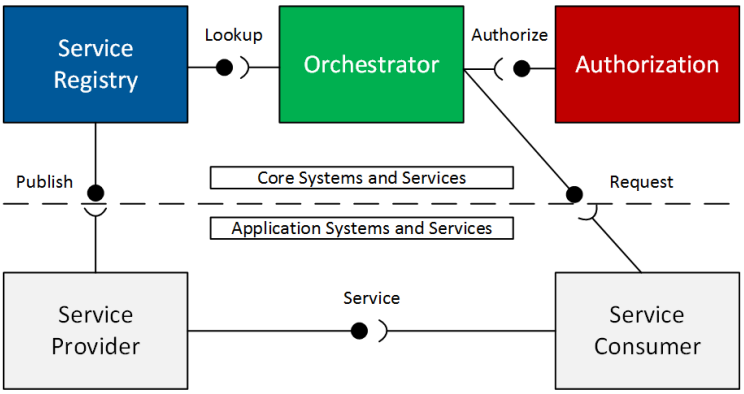

Figure 3.3. *Service discovery in Arrowhead Framework [5].*

In figure 3.3, the co-operation between the core and the application systems is presented. The Provider system registers its service to the service registry, from which the consuming system can discover it via consulting the orchestrator. Before the orchestrator responds to the requesting consumer, it makes sure that the consumer is allowed to use the service, by consulting the authorization system.

3.1.3 Supporting Core Systems

The supporting core systems are extensions to the core systems, which provide either an infrastructurally significant functionality or so commonly needed set of services, that if

not provided officially, application system developers would separately end up developing their own versions of. Unlike the core services, the supporting core services are not mandatory.

On this section, only the supporting systems available in the current version of the framework are introduced. However, the GitHub repository of the framework has multiple feature-branches for upcoming supporting core systems [4].

Event Handler

The event handler provides means for event passing between systems. It acts as a dispatcher between event publishers and event subscribers. Systems can introduce themselves as subscribers of a particular event, and once another system fires the event, it is passed to the subscriber by the event handler [57].

The event handler can filter events based on rules set during a subscriber registration. The rules can be arbitrary key-value pairs, and they are stored in the MySQL database as are the subscriptions.

Gatekeeper and Gateway

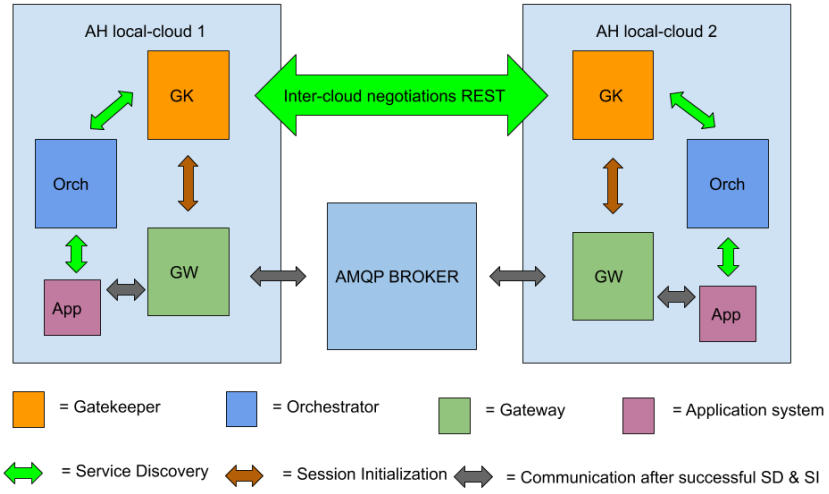

Figure 3.4. *Phases of global service discovery [4].*

In figure 3.4, the phases of global service discovery and establishing a tunnel between two local clouds is presented. This mechanism allows interaction between systems in multiple different local clouds [4].

The gatekeeper system is used in service discovery, between application systems under different local clouds. If the service request received by the local orchestrator is configured in a way which allows the global discovery, and authorization system is aware of neighbouring clouds, the request is relayed to the other core systems' orchestrators via the gatekeeper system. If the discovery is successful, an intercloud connection is formed

between the systems in different local clouds[4].

The gateway system is responsible for the tunnelling between systems under different local clouds. After a successful global service discovery, the gateway acts as a proxy, and on both ends of the tunnel, the systems involved do not know who they, in reality, are interacting. Instead, the address of the local gateway system, and a port reserved for this particular session is received from the orchestrator by the consumer. From the provider's perspective, the requests at runtime are seemingly coming from their local gateway system [4].

While the communication between the core systems and application systems is happening with REST, the tunnel established between the gateways uses broker as means for communication. One example of a supported broker is an AMQP broker known as the RabbitMQ [50]. In other words, in the case that AMQP broker is used, the gateway system is an AMQP client that offers a REST interface to the outside world for application systems in the local cloud it resides in [4].

3.2 Supporting Tools

To be able to evaluate the Arrowhead Framework, a set of external tooling for supporting the Arrowhead local clouds is needed. Since the Arrowhead Framework itself does not provide means for persistent storage, deploying the containers and a programming environment, tooling in these areas are needed.

Aside of the needs set by the framework, other reasoning behind the selection of the supporting tools includes:

- The tool should be open-source.
- The tool should be in common use.

Based on both, the needs set by the framework, the goals introduced above and the conclusions made in the literature review in chapter 2, the presistent storage was decided to be built around PostgreSQL[46], the deployment around Docker [14] and the application system development around Node.js [42].

All the techniques have alternatives that could meet the criteria. In case of Node.js and Docker the main argument against competing technologies was the wide use of chosen techniques. Although, especially in Node.js's case, environments like Python [49] and various JVM [30] based languages, would have done the job as well. The main argument in the case of persistent storage was the fact that PostgreSQL based solutions are in wide use at Metso. Alternatives for persistent storage would have existed as well. Especially NoSQL based database solutions like MongoDB [38] and CouchDB [12] would have been able to do the job. In the following sections the supporting tools are introduced in more detail.

3.2.1 Persistent Storage

The states of various resources offered through services of Arrowhead application systems need to be stored. However, since the framework is not offering any supporting core systems to help in achieving this, other solutions are needed. On this section, one possible set of tools for data management are introduced. All the tools are widely used in industry and available through open-source licences.

PostgreSQL

PostgreSQL is an open-source object-relational database engine. The project is based on the POSTGRES project initiated in 1986 at the University of California at Berkeley. With millions of users, it is currently one of the most commonly used database systems in the world [46].

A PostgreSQL specific dialect of SQL-language is used while interacting with the database management system (DBMS). Like in many other relational database systems user-defined data-types, views and functions are supported [46].

On top of support for primitive data-types like integers, doubles, string and booleans, PostgreSQL also natively supports document data-types in JSON/JSONB, XML and key-value form. Also, unlike most SQL databases, Postgre also has native support for array data type [46].

PostgreSQL Extensions

PostgreSQL provides a wide variety of means for extending its functionality. On top of function declaration in SQL, which is commonly available in any modern SQL database system, PostgreSQL enables dynamically loadable functions and types written in C [46].

For extension development in C, set of headers are provided. These define the standard interface for user written code, and a set of PostgreSQL specific types, macros and functions. For example, memory management is done via `palloc()`, and `pfree()` functions, instead of standard `malloc()` and `free()` [46].

The modules written in C, are compiled as shared objects and can be loaded on runtime, by introducing an SQL function similarly like one would introduce a function implemented in SQL. The only difference is that on the place of SQL clauses implementing the functions "body", the location and name of the compiled module are specified [46].

User is also able to install so-called "procedural languages". These packages are effectively interpreters as extensions, and allow functions to be written in some commonly known scripting languages, like Python or Perl. If need be, the user is also able to develop a procedural language package for domain-specific ad hoc programming language and extend the database system with it [46].

Timescale Database

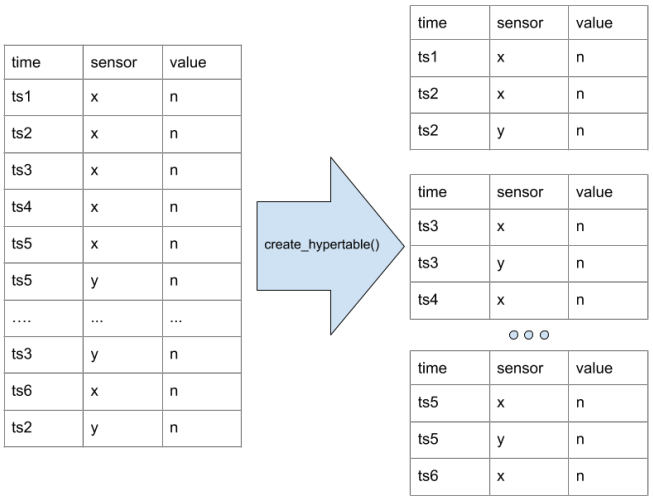

Figure 3.5. *The chunking is done with the* `create_hypertable()` *function [55].*

Timescale database (TSDB) is an extension for PostgreSQL which allows its usage as a time-series database. The reformation is achieved via an extra layer of abstraction known as the hypertable, which in turn is based on a concept called "chunking" [55].

Since TSDB is built on top of PostgreSQL, the user is able to perform queries and insertions with standard SQL and create views[1] and indexes on its hypertables. The most apparent difference compared to the traditional use of PostgreSQL is the way how tables are created. To create a TSDB hypertable, one has to create a regular table with a timestamp field. After the table is created, a function for reformatting it to hypertable is used [55].

The problem that chunking tries to solve is the overhead caused by the way how PostgreSQL and many other relational database systems store data Internally. PostgreSQL indexes stored data by using b-trees, with tree per table principle. The point behind this is to make access to data fast [55].

However, if the size of the table grows too big, the tree will not fit in RAM, which will trigger the swapping mechanism, and some memory pages containing the trees data are moved to disk. Since disk reads are much slower than reads from RAM, — or on par with turtles if compared to reads from CPU caches — this will reduce the performance of operations run on the table. This is bad, especially in case of time series data, where most recent data is needed most commonly, and random searches on the whole table are infrequent [55].

Figure 3.5 presents the way how `create_hypertable()` chunks the regular table into a hypertable.

[1] function for creating a continuous aggregate view is also provided

The default way is to chunk based on time intervals, but the time field can be of any incrementable type. Under the hood, chunks are also tables, which means that each chunk will have its own b-tree for indexing. This allows the b-trees of latest chunks[2] to be kept entirely on the RAM — and partly even on CPU caches — which in turn makes access to most recent data constant time and independent on the size of the whole hypertable [55].

The extension is also capable of chunking the table with an extra field. In IoT use case, for example, one might chunk not only by the standard timestamp but also by the name of the sensor. This way, the TSDB tries to keep data from different sensors in different chunks. Which in some use cases, should improve the performance [55].

PostgREST

PostgREST is an open-source project, which provides means for accessing PostgreSQL database via a REST interface, which is generated by the PostgREST automatically, based on the schema of the database [47].

PostgREST exposes the database as a set of resources which are mapped to tables and views with URIs. On top of basic CRUD-operations on these URIs, the user can also perform complex queries with postgREST specific syntax, where the query is passed as parameters. Body of the message is passed in JSON form, in which each row queried, updated, inserted or deleted, is represented with an object [47].

3.2.2 JavaScript and Node.js

JavaScript was originally designed — and is still mainly used — as a scripting language for the browser. The language is defined in ECMAscript-standard [16], which has multiple implementations, mainly by major browser vendors. Most of the modern implementations of the standard use so-called Just-In-Time-Compilation, where opposed to traditional interpreters, the code is not mapped to machine instructions via interpreting, but compiling the code "Just in time", before the execution, which enables various optimization schemes since the compiler can modify the output according to the state of the program runtime [42].

Node.js is the most known JavaScript environment outside the browsers. It is an asynchronous runtime based on Googles V8 engine, initially developed for Chrome Browser. Node.js was designed for development of I/O bound applications, for example, HTTP-servers. Nodes execution model is based on single-threaded event-loop, which heavily utilizes the operating systems non-blocking I/O-event mechanisms, to gain the ability to run tasks concurrently [42]. Node.js has implementations for all major operating system platforms. On Linux, its asynchronous execution is based on a bundle of system-calls

[2]TSDB has a mechanism for keeping the chunks evenly sized, this is not covered in this brief review, more information on this can be found on their web page [55]

know as epoll [18].

Used Libraries

Node.js comes with a package manager know as NPM [35]. With NPM, users can extend their application, with modules written by other users organizations and companies. Sharing open-source modules in NPM is common among the community.

Most relevant modules used in the demo application include:

- Express — an HTTP server library. Express itself offers relatively little functionality. Instead, it offers a clear interface for creating HTTP-servers, by extending its main object commonly known as the "app", with functions known as the "middleware" [20].
- Axios — an HTTP client library, which has a simple interface based on promises. It also offers other functionality like automatic parsing of responses to JSON format [6].
- Node-OPCUA — a library for creating OPC-UA servers and clients [41].

3.2.3 Docker

Docker is a platform which provides a set of tools for container management. Docker offers an abstraction called image, that can be used to initiate containers which in Docker's context are runtime instances of an image. Images can be stored in the so-called registry, of which the most commonly known is the public docker hub [14].

Images have a layered structure, and each command in a so-called docker-file, which is a YAML-file that defines the build process of a particular image, adds a new layer. For example, the first row in the docker-file usually provides so-called base image, on top of which user can add new layers like folders containing the application code, by issuing a copy command, or shell commands that should be run on the ready image when it is spawned as a container, by issuing a run command. This structure allows only the layers that change to be built in case of a rebuild [14].

The layered structure also makes extending of any given image possible. The base image refers to an image which has not yet been extended and only includes the layer it itself represents. However, the images that are extended from the base image and contain multiple layers can be further extended in separate build process defined in docker-file, which defines the multi-layered image as its "base image" [14].

The building, deploying, undeploying, and docker registry pushes and pulls are done via so-called docker client, which offers a command-line interface for management. The commands issued from the client are handled by so-called docker daemon also commonly known as the docker engine, which is the central piece of Docker and responsible on handling the needed chores under the hood [14].

Docker-compose

Docker-compose provides means for starting and stopping containers as a bundle. The configuration of the bundle is specified in so-called compose-file. Docker-compose allows multi-container setups that are easy to move to other environments while keeping the setup the same [14].

Docker-compose is mainly used in development environments, due to the easy interface, which allows interacting with the whole system with one command. Bundling eases the development process, since starting of applications is simplified drastically, compared to the more traditional way of starting all dependencies individually or via ad-hoc scripts. The interface of Docker-compose is similar to "plain" Docker, and it contains, for example, commands for starting, stopping and logging the standard streams of containers. Docker-compose also eases the creation of DNS enabled virtual networks [14].

4 METHODOLOGIES

In this section the methodology used to in solving the research questions is presented. The primary method is to develop a demo application which aims to achieve the features of an ideal system, which are introduced.

Secondly, the test setup at research facilities at Metso's Tampere factory is introduced. The evaluation of the software is done with a setup including a vibrating screen exciter, wireless vibration sensor boxes and an industrial-scale computer that serves as the edge device, which is communicating with a private cloud.

4.1 Approach for Solving the Research Questions

The research questions are tackled from perspective of design science (DS). DS is a scientific problem solving process that concentrates on the study of the artificial, instead of more traditional study of the natural. The goal of DS is to produce artefacts and it is used especially in the field of information system(IS) research [28].

Hevner et al. summarizes the role of the design science research in form of two fundamental questions that need to be addressed [28]:

- "What utility does the created artefact provide?"
- "What demonstrates that utility?"

The artefacts can be instantiations, constructs, models or methods, and they must either solve a relevant problem in a novel way, improve an existing solution or provide a more effective alternative for the state-of-the-art solutions. The objectives of the artefact must be set, and based on the objectives, the artefact must be created and its effectiveness must be proven, while the whole process is documented. Depending on context of the project, the effectiveness can mean things like: functionality, completeness, consistency, accuracy or reliability [28].

The model chosen as the methodological framework for this thesis is design science research process (DSRP) by Peffers et al. [44]. It aims to offer a structured model for DS based research in the field of information systems by providing a process for practising it, a mental model on what the output of DS research should look like, while being consistent with processes in other fields of study [44]. The six activities of the research process are presented in figure 4.1, and explained in more detail below.

- 1. Problem identification and motivation — research problem is defined and the value of a solution to the problem is justified [44].
- 2. Objectives of a solution — The objectives for the solution are derived from the research problem and introduced in a detailed fashion. Objectives can be quantitative or qualitative, and they may be presented through needed improvements to an existing artefact or a set of goals to a problem that has not yet been solved [44].
- 3. Design and development — The artefact is first designed and then, based on desired functionality, created. The artefact can be a method, model or a construct, each of the terms enlisted are used in broad sense [44].
- 4. Demonstration — The artefact's capability to solve the research problem is demonstrated. This can happen in form of an experimentation, a case study or other scientifically sound method [44].
- 5. Evaluation — The results of the demonstration are observed and evaluated in terms of objectives set in activity 2. After the evaluation, feedback to activities 2 and 3 is given, and possible reiterations are taken to improve the artefact and/or objectives [44].
- 6. Communication — The process is documented, possible future work in form of reiterations is suggested and then communicated through appropriate channels such as professional and/or research publications [44].

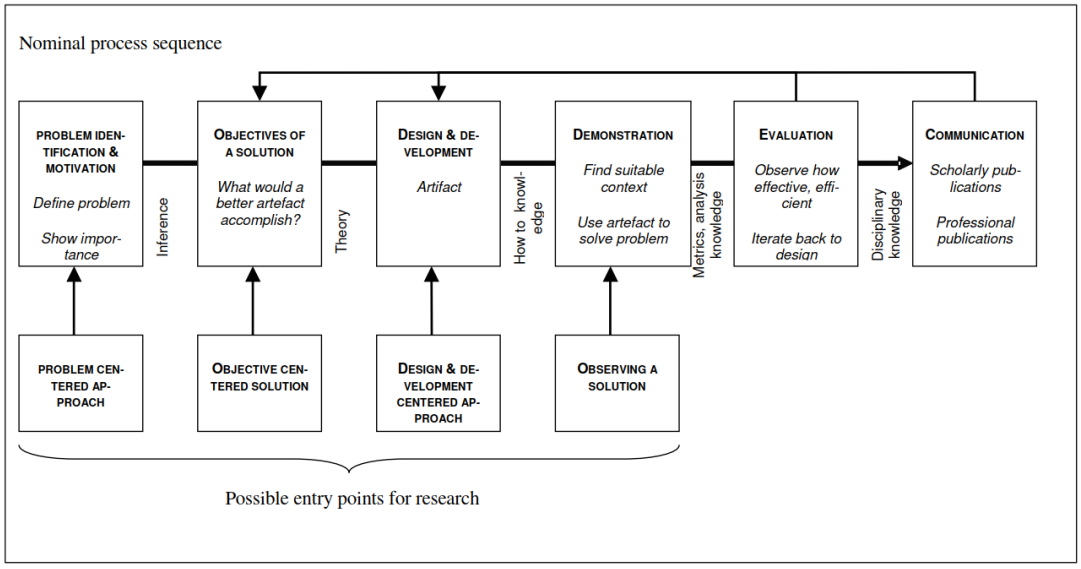

Figure 4.1. *Design science research process (DSRP) defines a structured model for design science [44].*

Since it is possible that either the problem definition, objectives, design or even the artefact itself exists when the research process is started, the authors of DSRP enable four possible entry points for the research [44]. Since the Arrowhead Framework is a "ready" solution, the research process used in thesis starts from the "entry point for observing a solution".

In the demonstration phase of the DSRP a set of functionalities, which would be available

in an ideal system, are presented. The goal of the development of the demo application is to achieve the functionality of an ideal system, which is defined below. The primary way how research questions represented in section 4.1.1 are answered comes from the evaluation on how well did the Arrowhead Framework introduced in section 3.1 achieve the requirements set.

During the development process the quality, design and usability of the Arrowhead Framework are taken into account, and reported in the evaluation section 6. The main point of view in this report is on the effects that the functional caveats, bugs or implementation details have on getting to the ideal system. Some possible solutions to problems found are presented as a feedback for next development and design iterations of the framework.

4.1.1 Ideal System and Expectations Placed to AHF

The ideal system is defined in the context of the research questions:

- How can Arrowhead Framework help in configuration of the data flow from edge to cloud?
- How can Arrowhead Framework ease the installation of new edge devices?

In other words, what is meant by: "configuration of the data flow from edge to cloud" and "ease of installation of new edge devices", is defined in this section. Since the Arrowhead Framework, like any other tool, does not solve all the problems below the sun, the definition of the ideal system tries only to take into account things that Arrowhead should be able to handle based on the review presented in section 3.1.

However, while the fact that the artefact, the Arrowhead Framework, is not put in to place where it is not designed to be used at, it is also important to keep the real-world problem, presented in introduction chapter 1.2, at mind. In practice; the usability, reliability and completeness of the Arrowhead Framework are taken into consideration. In other words, if the ideal system is achieved, but the framework does not, in reality, make things easier, but instead only moves hard things into an Arrowhead specific "abstract bubble", it is not, in reality, solving the problem.

Configuration of the Data Flow From Edge to Cloud

- 1. The configuration of what data is pushed from the edge to the cloud should be definable.
- 2. The interval of pushes should be definable.
- 3. The services implementing the functionality should be easily changeable, both in the edge and in the cloud.
- 4. The services at the edge or at the cloud should be discoverable by systems that are authorized to consume them.

In practice the item 4 summarizes the expectations set on the Arrowhead Framework, the service discovery functionality of the Arrowhead Framework is at its core and it is expected to be able to help in getting to the ideal systems items 1,2 and 3. For example, in case of items 1 and 2 the service that offers the interface for configuration should be discoverable. In case of item 3 the service discoverability and especially the late binding that it allows should help.

Ease of Installation of new Edge Devices

- 5. Installation of new edge computer should happen with minimal need for manual configuration. To achieve this, the following must be true :
    - 5.1. The configuration of the edge computer must be pullable over the internet.
    - 5.2. The stock OS-image of the computer must be able to pull the configuration.
    - 5.3. Fall-back plan in case of simultaneous network and device malfunction.

In practice, the subitems of item 5 should benefit from the functionality offered by the Arrowhead Framework. Similarly to items from 1 to 4, service discovery plays a big role. Additionally, in case of the subitem 5.3, the concept of interconnected local clouds should help, in theory, if the service discovery is functioning as it should, backup Arrowhead local clouds that exist in the edge could be used in case of network malfunction between the edge and the cloud.

It is also important to note that the creation of an OS-image that is capable of starting the Arrowhead core systems and the systems responsible on fetching the configuration is left out of the scope of this thesis. This is mainly due to the fact that commonly known mainstream projects like systemd [54] exist and if the Arrowhead system of systems can be started while the operating system is already running, they can also be configured to start while the computer is booting up. Therefore, it is safe to assume, that providing a solution that solves the items listed above and in section 4.1.1 proves the fact that such OS-image could be created.

4.2 The Setup Used for Evaluation

One example of a vibrating screen manufactured by Metso is given in figure 4.2. The vibrating screen is used to separate materials, with different granularities form each other. Vibrating screens are most commonly used in mining and aggregate industries.

The vibrating screen used in the evaluation is an older model, and its primary use case is to offer a platform on which the research team at Metso Minerals Tampere factory can examine how materials sent by customers act with different setups. As a secondary, the screen is used as a test platform for a vibration-based condition monitoring system, and it is instrumented with sensor boxes that collect the vibration data from the screen.

The sensor boxes are equipped with Bluetooth Low Energy radio and pendulum based

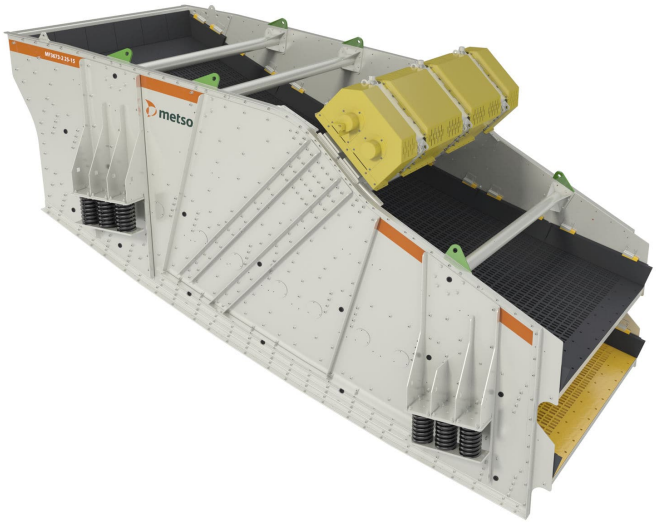

Figure 4.2. *One example of a vibrating screen manufactured by Metso.*

energy harvesting mechanism, that collects the energy needed by the electronics from the movement of the screen. Therefore, the sensor boxes are fully wireless.

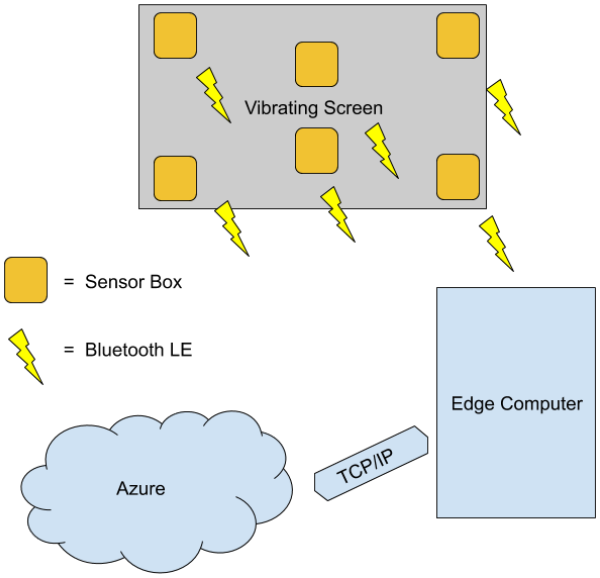

Figure 4.3. *Overview of the setup used for evaluation.*

A general view of the setup is presented in figure 4.3. The edge computer lies in an electrical cabinet within proximity of the screen, and it is equipped with Bluetooth Low Energy receivers, which listen for the radios of the sensor boxes and collect the raw vibration data. The edge computer is an Industrial scale x86 computer, and it has a server flavour of Fedora Linux running on top of it. The data collected at the edge is sent to a virtual machine in the Microsoft Azure cloud.

5 IMPLEMENTATION

At first the software stack of the implementation and the roles of the tools introduced in section 3 are presented. Secondly, the method used for application system development is introduced. The application systems are all containerized JavaScript applications, which are built with two-staged build process automated with Python script.

After the software stack and the development method is introduced, the application systems that implement the business logic of the Arrowhead system of systems are documented. First, the implementation is presented, and finally, alternatives are speculated.

5.1 The Software Stack of the Demo application

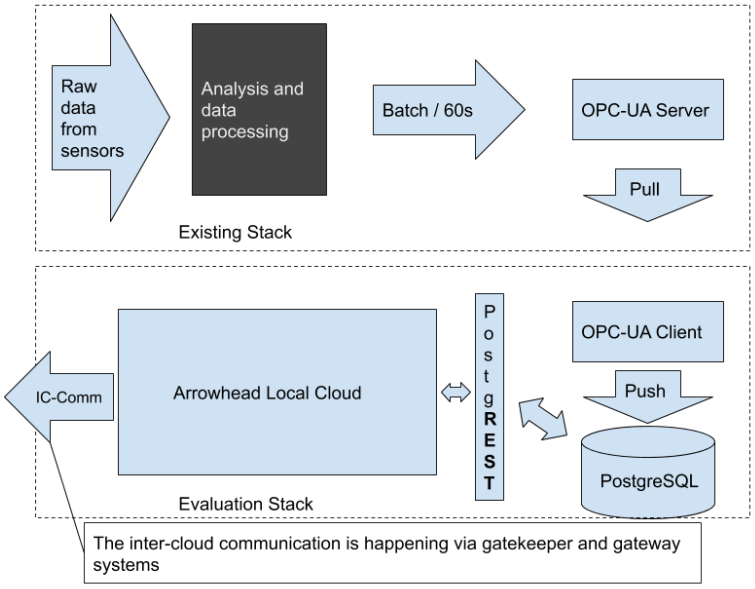

Figure 5.1. *The software stack of the edge computer.*

In figure 5.1 the software stack relevant to the demo application, running on the edge computer is shown. The existing stack partition in the picture refers to parts that are already present in the current system.

The analysis and data-processing "black-box" is responsible for the actual condition monitoring chores presented in section 2.2.1 performed at the edge. The "black-box" takes care of converting the raw vibration data collected from the monitored equipment and makes a new batch of data related to the condition of the machine available at predefined

intervals, currently 60 seconds. The batch can be fetched from the OPC-UA server, which also serves the "upstream" of the current system, not visible in the figure.

The evaluation stack partition includes software installed for the demo application. The Data available in the OPC-UA server is pulled to a PostgreSQL database with an OPC-UA client implemented in JavaScript. The client listens to the server for events and pulls the data once a new batch becomes available.

The tables and the views in the PostgreSQL database are made available as REST services by the PostgREST. The goal while implementing the schema of the database is to make every commonly needed database operation available as a view or stored procedure. This approach allows simple REST-based queries and insertions for Arrowhead Application systems.

The software stack installed on the Azure virtual machine is presented in figure 5.2. The stack is similar to the one on the edge. The PostgreSQL database on the cloud side acts as the final destination of the data from the demo applications perspective. However, unlike the vanilla installation of PostgreSQL on the edge, the cloud installation is extended with Timescale-DB, which allows higher throughput of condition monitoring data.

Both the edge computer and the cloud have one Arrowhead local cloud instance with application systems that can communicate with systems in another local cloud, through the Arrowhead gateway and gatekeeper systems. The AMQP broker needed by the gateway systems is hosted on the Azure VM.

All the components are run as Docker containers, including the core systems and application systems in the Arrowhead local cloud. The Docker images are stored in an image repository, where they are pushed from the laptop used in the development.

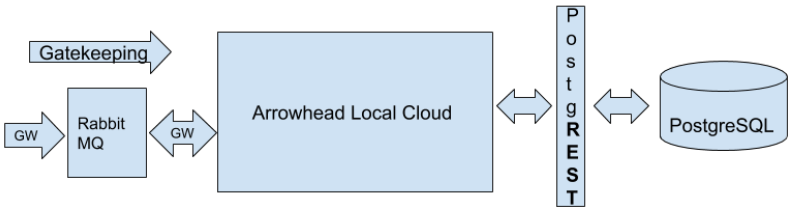

Figure 5.2. *The software stack of the Azure virtual machine.*

5.1.1 Mocking of the Edge

Since vibrating screens are loud and violent machines, and the instrumentation mounted to the vibrating screen does not produce any condition monitoring data when the screen is not running, alternative to running the software on the physical edge computer must be used while developing the demo application.

The OPC-UA server used as the data source of the demo application is therefore mocked

on a virtual machine running on the laptop used for development. The mock-server is implemented with JavaScript, and it produces random data which is available through a similar interface compared to an actual physical edge computer.

5.2 Application System Development

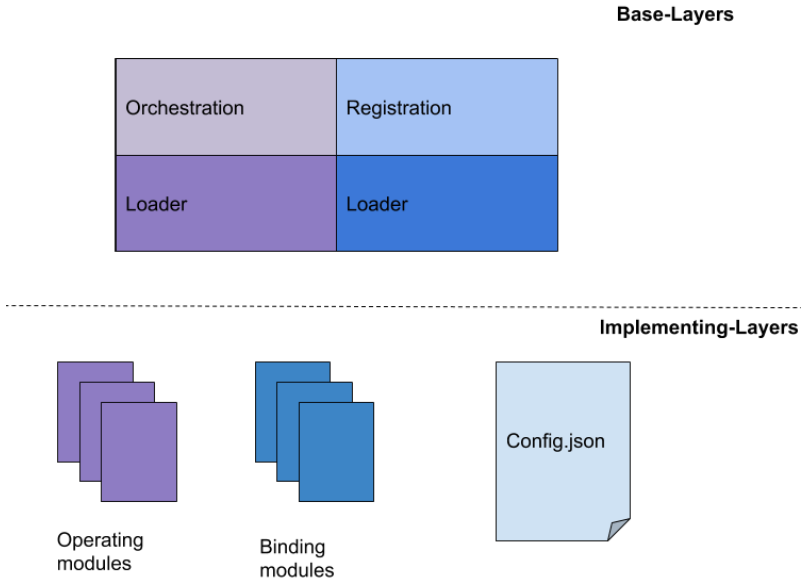

Figure 5.3. *Insides of a Docker image of an application system.*

The basic structure of an application system is presented in figure 5.3. All application systems in the demo application are Docker images, which are built with a two-staged build process. In the first stage, a Docker image for the base layers is built. The base layers are capable of loading modules, registering and deregistering services implemented in the modules and issuing service requests for the orchestration system based on needs of the modules.

On top of the Docker image containing the base layers, an implementational layer is built in the second stage of the build process, presented in figure 5.4. The build process is automated with a python script that takes an input with folder structure with all the available config files, each representing one application system, a collection of reusable modules and a Docker file that specifies the build process of the final Docker image.

The following steps are included in the process:

- Step 1. A temporary folder structure is created in /tmp.
- Step 2. A config file is loaded from the input folder structure and copied to /tmp.
- Step 3. All the modules specified in the config file are copied to the folder structure in /tmp.
- Step 4. The build process of the image is started based on the Docker file. This includes a step where the modules and the config file previously copied in the /tmp

are copied inside the Docker image.

- Step 5. The Docker image is tagged with the name specified in the config file and pushed to the Docker repository.
- Step 6. If more config files exist, the process continues from step 2, if the current one was the last one, the temporary folder structure is deleted, and the process ends.

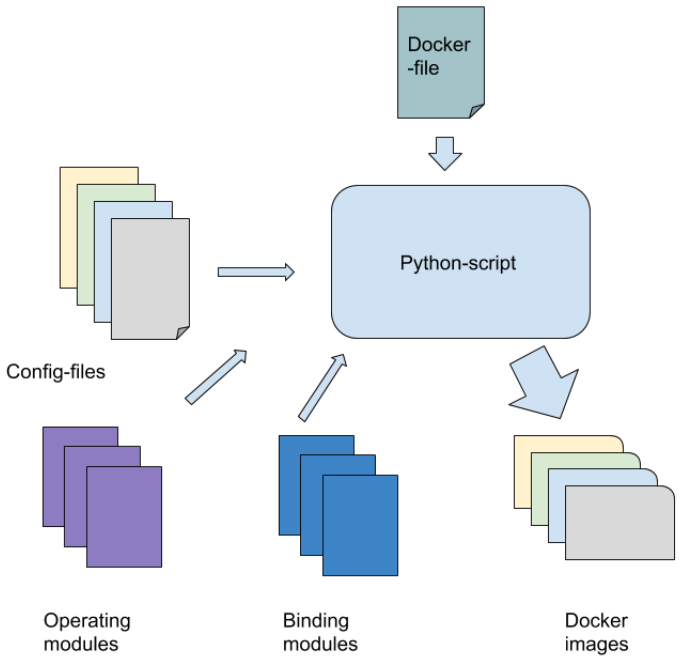

Figure 5.4. *The build process of the application system Docker images.*

5.2.1 Modules

As stated above, in the brief introduction to the build process, the image containing the base layers can load the modules at runtime. The configuration-file loaded inside the image, plays a huge role in this process. The structure of the configuration-file is presented in figure 5.5.

The modules implementing the functionality of the application system are separated into two categories; ones that implement operations, and ones that use operations implemented by the former. The modules that implement operations are referred to as operating modules, and the modules that use operations are referred to as binding modules. At start-up, the loader loads all modules that are inside a folder, inside the container, specific to the module type.

The loading process of operating modules includes a validation phase where the loader makes sure that modules implement operations that the configuration-file says they are implementing. After the validation phase, operations marked as used, in the sections describing binding modules, are passed to the correct module within a single object, from

```
{
    "cloud_name": "cloud",
    "cloud_operator":"metso",
    "system_name":"datacloud",
    "system_port": 3015,

    "modules": {
        "binding":[
            {
                "name": "data",
                "providesService":"batch",
                "uses":["pushToDb"]
            }
        ],
        "operating":[
            {
                "name" : "batch",
                "consumesService":null,
                "offers":["pushToDb"]
            }
        ]
    }
}
```

Figure 5.5. *One configuration-file represents one application system, and it is used both during the build process, and during the runtime.*

which the binding module can use them during the runtime, without the need to know what operating module is implementing the operation. This allows re-usability of both types since modules, using and offering operations, can be changed independently of each other.

From Arrowheads perspective, the operating modules consume Arrowhead services, and the binding modules provide Arrowhead services. However, neither providing or consuming is mandatory. In case of operating modules, for example, one is free to implement operations that do not consume any service that needs Arrowhead specific orchestration, for example in cases where the location of the service is already known, and it is not going to change in the future, there is no point in re-discovering.

One module of either type is restricted to consume or provide a maximum of one Arrowhead service, although multiple instances of the same service can be consumed. If one application system is consuming or providing multiple services, multiple modules are needed. The number of modules in one application system is not limited to any number. However, a large number of modules is probably a sign of a need of refactoring to multiple application systems by creating a new configuration-files, that are reusing the modules that were used in the large one.

The Module Interfaces

Both types of modules have to require an interface specific to their type. This way, the loader system can load them properly. The JavaScript files for the interfaces are included in the base-layers.

```
const moduleInterface = require('../../lib/common/bindingModule')
```
(5.1)

The interface object of the binding module is included as presented in (5.1).

```
const moduleInterface = require("../../lib/common/operatingModule");
```
(5.2)

The interface object of the operating module is included as presented in (5.2).

```
const moduleObject = moduleInterface.init(module);
```
(5.3)

After the modules interface-object is included, the init member-function presented in (5.3) needs to be called with the node.js specific module-object presenting the file as its sole parameter. The module-object is used to identify the module during the loading process.

In case of operating modules, the init-function returns an object with two functions, one for getting the address of the consumed services and one for flushing the address cache of the orchestrator module. The caching mechanism prevents the un-needed calls to the orchestrator core system. Neither of the functions needs any parameters. The orchestrator library already knows what service the operating module is consuming; since the loader passes the information needed from the configuration file.

All the operations that are marked as "offered" in the configuration file need to be present in the module.exports-object of the operating module. This way, the loader can find them and pass them to the correct binding module.

In the case of binding modules, the init-function returns all the operations used by the module within a single object. If the module is providing a service, the URI, that the module should listen, is also within the returned object.

If the binding module provides services, the express-app-object needs to be exported in a standard JavaScript way. The exported app object is merged to the application systems main express app object, by leveraging the middleware functionality offered by Express.

A Side Note on the Module System

Besides the building of Docker containers and easing reuse, initially one additional goal of the module system was to ease the integration of servers and client SDK's generated

from OpenAPI documents [43]. To some extent, this was successful, and one test implementation even had all the express servers in it generated from OpenAPI 3 with swagger-node-codegen [53]. The servers in the test implementation were altered by hand to be fit binding modules and successfully built to use handwritten operating modules.

However, it soon became apparent that in the case of the demo application, that is introduced in the next section, just writing the module code was a better choice. After all, writing server code with libraries like Express is already a quite optimized procedure in terms of keeping the boilerplate at minimum. The extra work that was needed by the generation approach was mostly due to the manual alteration needed for the output and other additional tasks, like writing and keeping the OpenAPI document up to date with the manually altered code.

While it did not make sense to use a generator in the demo application, some more demanding setup in the future might benefit from generation possibilities that OpenAPI offers. With slight alterations to some existing open-source generators, like swagger-node-codegen [53] or express-openapi [21] all the binding modules that provide services could technically be generated directly from OpenAPI documents.

Also, the OpenAPI generated Client SDK's could ease the writing of operating modules that consume services. Especially the cases where the services have a large number of sub-resources, a lot of boilerplate and error-prone document validation code is typically needed before they can be used.

5.3 The System of Systems

The structure of the system of systems is presented In figure 5.6. The application systems are separated into two groups; systems responsible for the data-path, through which the condition monitoring data is transferred, and systems responsible for the control-path, which are used in configuring the local clouds at the edge, including the way how the data-path is used.

The data-path has two Arrowhead application systems, one at both local clouds. The DataSystem on the edge-computer is pushing data to the cloud, through the batch-service provided by the DataSystem.

The control-path also consists of two application systems, one on each local cloud. The remoteControl system at the local cloud in the Azure VM provides three services, of which the pollControll system consumes two at the edge computer. The third one is used for controlling the data-flow from the cloud.

All the services where the consumer is at different local cloud are discovered through Arrowheads gatekeeper systems introduced in section 3.1. However, since the Arrowhead gatekeepers services used for inter-cloud service-discovery needs to be visible to the consuming side, and on the local cloud, at the edge, this is not possible, only the services at the Azure VM can be discovered from the edge but not the other way around.

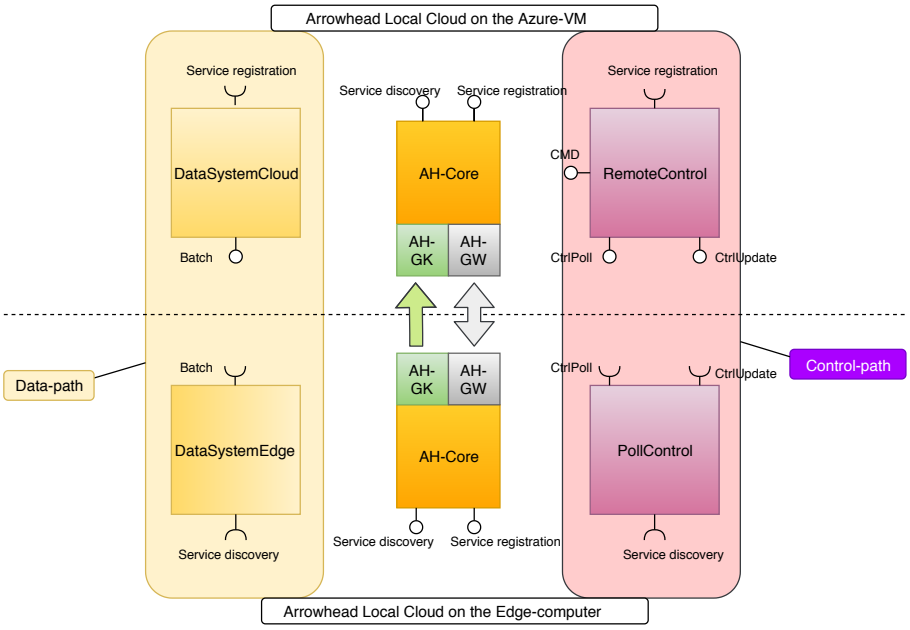

Figure 5.6. *Responsibilities of the application systems are separated into data-path and control-path. The service discovery and the service consumption between application systems on different local clouds is happening through the gatekeeper and gateway systems.*

5.3.1 Modules used in Application Systems

In this section the various modules used in implementing the application systems in data-path and control-path are introduced.

DataSystemCloud

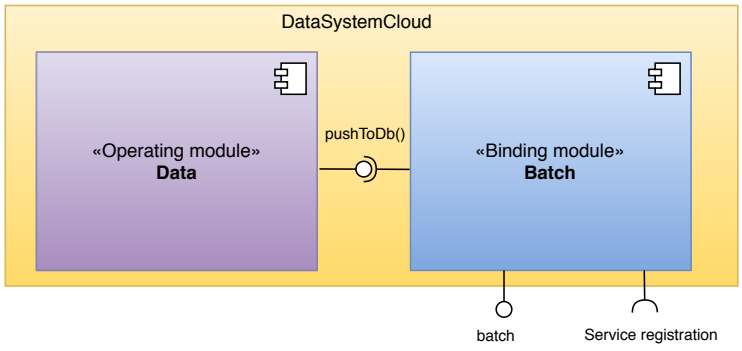

Figure 5.7. *Modules used in DataSystemCloud application system.*

The DataSystemCloud application system in figure 5.7 has two modules, one of each type. The binding module "Batch" provides similarly named service and uses one operation offered by the operating module "Data". As the name "pushToDb" suggests, the operation is used to push the incoming batch to the database.

Even though the demo application consists of only one edge-computer, the DataSystemCloud application system is compatible with multiple data sources. This enables schemes

where data from multiple monitored devices are sent to one virtual machine in the cloud.

Also, since the structure of the database in the Azure VM is not functionally dependent on the identity of the data sources. The Virtual Machine on the cloud, with all of its internals, could be duplicated. A good reason for duplication might, for example, be a heavy load, or latency issues caused by the distance between some portion of the edge and the cloud.

DataSystemEdge

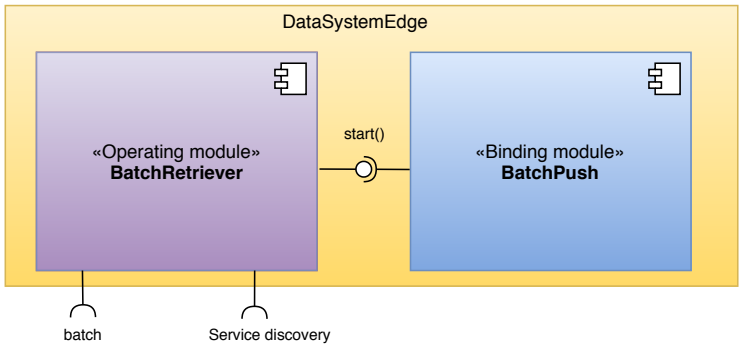

Figure 5.8. *Modules used in DataSystemEdge application system.*

The modules used in DataSystemEdge application system are presented in figure 5.8. Similarly to its counterpart presented above, DataSystemEdge also has one module of both types. The binding modules only purpose is to call start-operation offered by the BatchRetriever operating module.

The start operation starts the internals of the operating module, which is capable of sending a predefined batch with predefined interval to all providers offering the batch service, which are returned by the orchestrator core-system. The interval and a filter describing the batch, which is a subset of available data points stored at the edges database, has to be defined. If the orchestrator returns an address of a service provider, which does not have a proper configuration in the database, nothing gets sent to that particular provider.

Both the interval and the subset of data points can be altered during the runtime. This enables the reconfiguration of providers, including the providers that did not have any configuration when the orchestrator found them the first time.

By default, the name of the providers local cloud is used as the identifier, when the configurations are matched. However, more sophisticated schemes based for example on groups, which could represent multiple providers who need the same data, could be easily used as well, since the implementation of the application system already supports multiple providers per configuration.

It is also worthwhile to note that since the system can send data to multiple providers, the providers could as well be second or even third party stakeholders who have Arrowhead local clouds either at the edge or in their own cloud instance. This allows more sophisticated schemes where the batches sent to different parties could be filtered based on

interest, or permissions.

RemoteControl

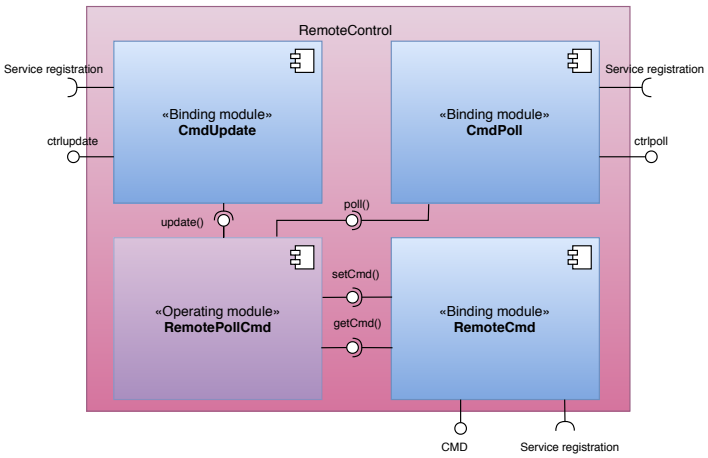

Figure 5.9. *Modules used in RemoteControl application system.*

The RemoteControl application system presented in figure 5.9 is implemented with four modules. Three of the modules are binding modules, and all of them provide a service. The only operating module within the application system — the RemotePollcmd module — is internally doing book-keeping, for commands sent through RemoteCmd modules cmd service. These commands are fetched via the CmdPoll modules poll service.

The commands can alter the batch and interval settings of any provider of "batch" service, and also a generic configuration object can be sent through the poller mechanism. Each edge-computer has its structure for incoming commands in the RemotePollCmd module. Only the latest command issued is stored, and successful fetch through the poll service erases the command structure of the poller, to avoid the fetching of already fetched command. However, since it is nice to know what was the last command issued, it can be fetched through the cmd service. This is achieved by using a boolean parameter at the URI when issuing a GET.

The ctrlupdate service provided by the CmdUpdate module is used by the edge-computers for notifying on changes in their configuration. The state of each edge-computer is stored inside the remotePollCmd module and can be fetched through RemoteCmd's cmd service.

The notifications that update the state can include data involving the configuration of intervals, batches and the generic configuration object that can be used for further configuration purposes at the edge. The updater can also send an array of available parameters through the ctrlupdate service.

Given that the edge-computers are consuming all the services provided by the RemoteControl application system in a sane manner, in which they poll the ctrlpoll in case of need for reconfiguration, send all the changes in configuration when they happen and now and

then update the available set, the cmd interface can be used to create user interfaces, or possibly even application systems that try to automatically "drive" the configuration of certain edge-computers to a particular predefined state.

PollControl

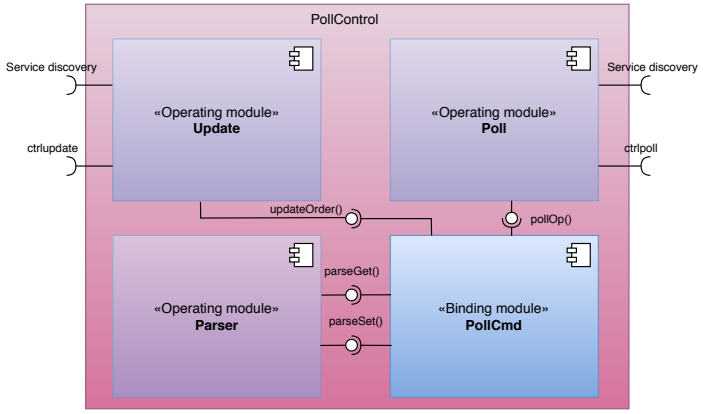

Figure 5.10. *Modules used in PollControl application system.*

The PollControl application systems presented in figure 5.10, is the counterpart of the RemoteControl application system presented above. The application system supports multiple points of control and can, therefore, be controlled by all the providers of the ctrlpoll service that the orchestrator core system is returning within its orchestration response.

PollControl has three operating modules. One of the modules consumes the ctrlpoll service, one consumes the ctrlupdate service, and one is responsible for parsing the commands received through the poll mechanism.

The binding module PollCmd bounds all the operations offered by the operating modules together in a way, where all the providers of ctrlpoll service get polled once in every 5 seconds. After a new command gets fetched, it is validated, and the state of the configuration is updated to the database by using the Parser-modules parseSet operation. In case of successful altering of the configuration, the new state is fetched from the database by using the Parser-modules parseGet operation. Afterwards, the state gets sent to all known providers of the ctrlupdate service.

The generic configuration object that is sent through the poller mechanism is used for configuring the analysis and data processing "black-box" at the edge-computer presented in figure 5.1. This allows schemes where the Arrowhead system of systems is the first thing that gets booted during an initial bootup process of the edge-computer. After the PollControl application system is up and it can fetch configuration of the "black box" from the Arrowhead local cloud at the Azure VM, and bootstrap the condition monitoring system that is feeding the data to the PostgreSQL database, which again gets sent as batches as is determined in the configuration.

5.4 An Alternative Control Approach

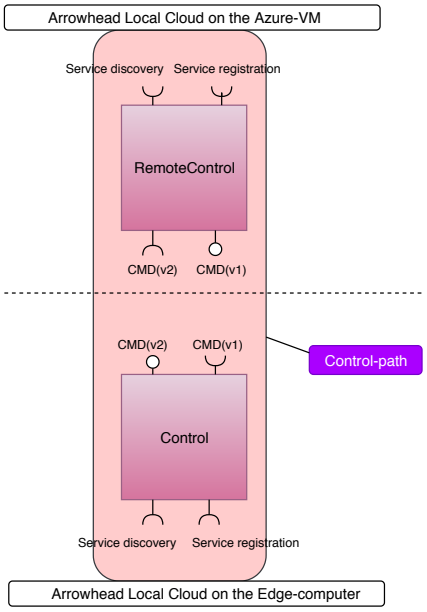

Figure 5.11. *An alternative implementation of the control-path.*

The reason behind the messy polling based control method lies in the fact that in the environment where the demo application is run on, the gatekeeper system in the edge is not visible to the gatekeeper in the local cloud at the Azure VM. However, if this kind of requirement was not placed or if the gatekeepers could see each other, the control-path could be simplified drastically, as is presented in figure 5.11.

In the alternative approach, the Remotecontrol application system directly consumes the cmd service offered at the local cloud of the edge computer. This way, the RemoteControl system becomes a mere proxy through which the resources holding the state of the configuration can be directly fetched and altered.

5.4.1 Modules used In the Alternative Control Approach

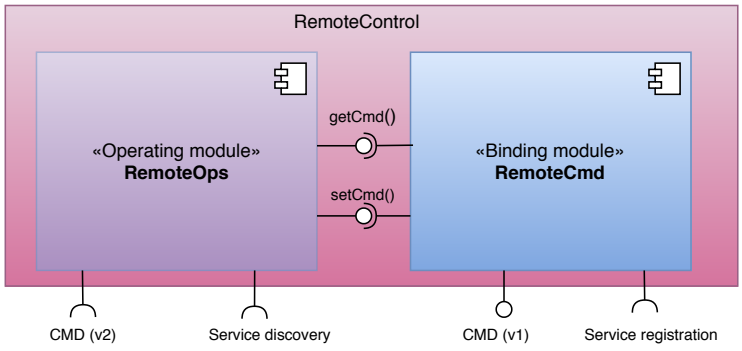

Figure 5.12. *Modules used in alternative RemoteControl application system.*

Since the module system allows reuse of the modules, the alternative RemoteControl systems presented in figure 5.12 is reusing the RemoteCmd module of the previous im-

plementation presented in figure 5.9.

The operating module of the new RemoteControl application system is simple. It offers the operations still relevant, from the previous implementations Pollcmd module, but instead of storing the commands issued, for edge computers to fetch via polling, it relays them directly to the cmd service provided by the control application system at the Arrowhead local cloud in the edge-computer.

Since the cmd services, consumed by the RemoteOps operating module are discovered through the orchestrator core system, only the providers that are found can be controlled through the RemoteControl application system.

On the Edges side, the new Control application system presented in figure 5.13 also leverages the parser module, which was originally developed for the use of the PollControl application system of the previous implementation of the control path introduced in figure 5.10. The parser module is slightly altered, and it uses the cmd service at the cloud. The reason behind this is that, since the polling mechanism is removed, cases where the edge computer wakes up after reinstallation or a boot-up, either unconfigured or misconfigured, the edge computer needs the means for fetching them, since polling from the cloud to the edge would be similarly messy.

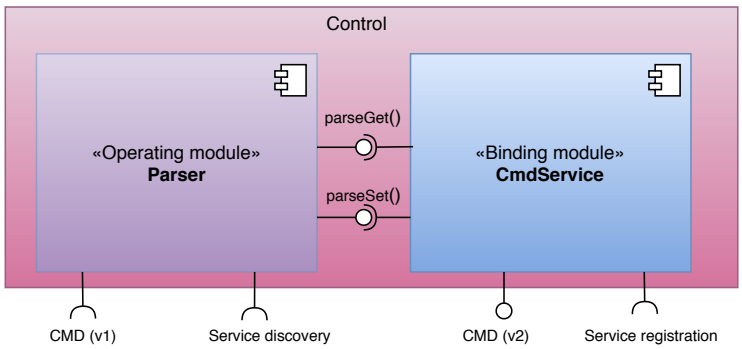

Figure 5.13. *Modules used in Control application system.*

6 EVALUATION

In this chapter, the validity of the Arrowhead Framework as the enabler of edge cloud architecture is evaluated. At first, the implementation is evaluated in the context of the ideal system defined in 4. Then upsides and caveats found during the development process are enumerated. Finally, the evaluation is summarized.

6.1 The Ideal System

As was expected in section 4, the service discovery mechanism of the Arrowhead Framework was successfully used to discover services implementing the needed features. The implementation introduced in section 5 worked as was planned, and all the items listed in section 4.1.1 were achieved, in relatively small experimentation with one edge computer and one virtual machine at the Microsoft Azure.

The configuration of the data pushes, and the edge computer was successfully fetched via service, that was discovered through the gatekeeper system. Both data and configuration were successfully moved through the tunnel established by the gateway systems. From this point of view, it can be stated that the system of systems implemented, achieved the features of the ideal system.

Additionally, the existence of multiple data sources and sinks were simulated with the existing local cloud instances by deploying multiple instances of existing application systems by using Docker's capability to deploy containers with ease. The simulation was successful, but it is important to note, that further experiments with a larger test setup would be needed for more definite answers. However, since a local cloud is a local cloud, the results should be similarly successful with a more extensive test setup with multiple local cloud instances on different machines both at the edge and the cloud.

6.1.1 Upsides of the Arrowhead Framework

Undisputedly there is potential in the systems of system approach taken by the Arrowhead Framework. This, combined with service-orientation, gives a clear mental model on how one would build things with the framework. The service discovery and authorization functionality offered by the core systems support each other well on a conceptual level.

The concept of the local cloud, the ability to discover services across local cloud bound-

aries via the gatekeeper system and the ability to consume the discovered services via the gateway system are all features that support each other well. Theoretically, a vast system of systems could be build, while the control of an individual system is still kept at the local cloud level.

Also, the way how gateway systems allow the application systems to use HTTP while in background broker technologies like AMQP, with denser data presentations handle the communication over the most significant bottleneck, the internet, is a feature worthy of mentioning. The way how the broker technology is used also brings consistency to application system development since the application systems only need the ability to understand HTTP.

6.1.2 Shortcomings of the Arrowhead Framework

While the version number 4.1.2 [1] of the evaluated framework suggests, that it is the fourth generation of a ready product, the maturity is far from the level of a ready product. Currently, the framework is only suitable for small scale test setups, like the one that it was used for in this thesis. This section goes into more depth on this front, and it can be considered as the feedback of the evaluation activity of this instance of DSRP.

Orchestrator

While basic service discovery through the orchestrator core system is available, it still lacks lots of features that would be necessary for it to be usable in the industry. One example of this is the infancy of the metadata-based service discovery. While registering services in the service registry, the application system can specify a set of arbitrary key-value based metadata-fields. However, the orchestrator system is not capable of fully leveraging this.

As an example, If the service has three metadata-fields which all describe the physical location of the system providing the service with different accuracy, one for the country, one for the city and one for the neighbourhood. The orchestration based on only one of these metadata fields is not possible in cases, where the consumer system wants to consume all services where the metadata-field is, for example, in a scope of a city. Instead, all the flags need to be matched. This means that, in this case, from the point of view of the orchestrator, the ability to add multiple metadata-fields is unnecessary, since only one metadata-field describing the neighbourhood would have a similarly bad result.

This is also the case if the services are registered with one metadata-flag per service, describing the country where the provider system is located. Now, If the application system wanted to issue an orchestration request with a list of acceptable countries for the provider system, this is not possible, which means that the application system is forced to make multiple service requests for the same service, each with one metadata-flag.

[1] standard MAJOR.MINOR.PATCH convention widely used in open-source projects is assumed

The lack of features in the orchestrator core system is a deal-breaker since it adds lots of pressure on the application system development. The application systems need to take care of the responsibilities of the orchestrator in all cases where the needs are not trivial. This results in convoluted code, where the service discovery calls, and the logic needed for parsing the responses, takes a larger role than the code needed for the business logic of that application system.

Authorization

Another problem in core systems involves with the authorization system, on which it is required to specify with one per system basis what system is able to consume which service. In figure 6.1, screenshots of the tables involved in this process are presented. The System table is used for storing information about core and application systems in Arrowhead local cloud, the global and local authorization tables are used to specify authorization rules in global discovery happening through gatekeeper system and local discovery happening through cores own orchestrator system.

As can be seen in the local authorization table, the column's "consumer_system_id" and "provider_system_id" refer to rows in the system table. Therefore, it is expected that the same amount of information is known from systems on both sides. On the system table, the only mandatory column is the port that the system listens to, which is irrelevant in the case of the consumer and HTTP clients in general. However, the name field of the system table is the one that is used when systems identity is determined.

The inconsistency on how the rows in the system table are set in the database is a small problem compared to the main problem that the current approach introduces. That is, of course, the fact that, before a consumer can discover anything, the authorization system needs to know precisely who that individual consumer is.

This strict policy means that the consumer and provider instances are tightly coupled to each other by the way how the authorization system is implemented. The strictness and tight coupling combined with the fact that the user has to figure out the deployment on their own, in the sense of actually starting the application systems, raises questions.

One of them is, that if the user has to couple the services in the database, by hand, and afterwards start them, why wouldn't the user skip the whole hassle of Arrowhead and couple the services on start-up, by providing the provider addresses and other stuff, like access-tokens in a configuration file or start-up parameters?

Another question raised by the tight coupling and the lack of dynamism caused by it is that is the Arrowhead Framework even providing a proper service discovery functionality or just a mere configuration hub of a sort? If so, how is the Arrowhead Framework going to compete against, for example, various tooling built around container technology, which already does not only offer means for configuring the "connections" and support for DNS but also provides means for deploying and starting the services [14]?

System Table:

id	address	authentication_info	port	system_name
9	0.0.0.0	NULL	8440	ORCHESTRATOR
10	0.0.0.0	NULL	8452	GATEWAY
11	0.0.0.0	NULL	8444	AUTHORIZATION
24	0.0.0.0	NULL	8454	EVENT_HANDLER
406	0.0.0.0	NULL	8448	gatekeeper
462	0.0.0.0	NULL	3015	datacloud
467	0.0.0.0	NULL	3013	remotecontrol
NULL	NULL	NULL	NULL	NULL

Global Authorization Table:

id	consumer_cloud_id	arrowhead_service_id
1	2	461
3	2	464
2	2	465
4	2	466
NULL	NULL	NULL

Local Authorization Table:

id	consumer_system_id	provider_system_id	arrowhead_service_id
1	2	1	2
2	1	2	1
NULL	NULL	NULL	NULL

Figure 6.1. *Authorization core system needs too detailed information about the consumer systems in the case of local service requests. On the other hand, in a global case, too much trust is given for the neighbouring local cloud.*

As can be seen in figure 6.1, global authorization takes a more relaxed point of view in terms of strictness of the authorization rules. On the global level, the authorization is done in groups formed by foreign local clouds themselves. This means that any application system from the specified foreign local cloud can consume the service specified on the row.

Most likely, the main reason for this more relaxed approach towards foreign consumers comes from the assumption that the local cloud that was authorized to consume has already taken care of the application system-level authorization. However, the evaluated implementation of the framework does not do that. Instead, the orchestrator at the foreign local cloud straight up goes and fires the intercloud service request without any further authorization processes.

Unarguably, an application system-level authorization of some sort is needed. In the case of local authorization, the current implementation is way too strict and demanding, and on the other hand, the global authorization does not exist at the system-level. This means that some further work on this front is needed. Some scheme that brought indirection by storing "authorization-tokens" instead of the detailed information about the authorized application systems themselves could offer a solution.

In this scheme, the authorization tokens could be added to the authorization systems database tables by the providing systems themselves, or by the user. Afterwards, the application system willing to discover a particular service would need to include a token that was associated with a provider or a group of providers inside its orchestration request. The orchestrator could then relay the token to the authorization system, which could verify the consumer's privileges. Ideally, the number of the tokens passed with an orchestration request could be larger than one, which would allow multiple providers associated with a different token to be passed in the response.

This scheme would allow the same kind of, although more flexible, group authorization as the current implementation of the global authorization has, since all the consuming application systems that have the token could discover the service, without a need for the authorization to know their identity on the level of addresses and system names. Also, the global authorization could use the same token-based scheme as the local one, since a token is a token independent of the place of its use.

Of course, this scheme comes with unanswered questions as well. How would the application systems get the tokens? How would the tokens get generated safely? Additional external tooling would probably be needed to solve the problems implied by the questions above.

Service Registry

Some problems exist on the level of interfacing between the core systems and the application systems. The most obvious example is found in the service registry, where the registration and the deregistration are not done in a REST fashion at all. I.e. the interface does not abstract the registry entries as resources. Instead, ad-hoc remote-procedure-call scheme on HTTP is used. For example, the deletion is done via an HTTP PUT on a "resource" with URI: "serviceregistry/delete".

Another problem in the service registry is the lack of support for a sub-resource management scheme. If a system provides a service, which has related sub-resources like its often the case when REST is used, the resource and its sub-resources can not be registered with one registration call. This means that as a solution either the sub-resources are registered individually, the systems consuming the resources "just have to know" what sub-resources exists, or the provider system itself offers a service for discovering the sub-resources, which can be used after the base resource is discovered successfully via core services.

At least in some cases, the individual registration of the sub-resources is probably a bad idea since URIs might be "deep", and they might have (multiple) variables in them. For example, what would be registered if the URI was "/machine/sensors/<sensorID>", where "sensorID" is a variable that is used to identify a particular sensor, and numerous sensors existed? Surely every possible id should not be registered in the registry as an individual service entry, especially if the authorization system controlling the access to the resource

is implemented as it currently is?

Outside small scale test setups, the assumption that the consumer "just knows" what sub-resources exists, is not ideal either since, in non-trivial cases, the amount of knowledge might become unbearable. Although, external tooling build around OpenAPI and their capability to generate SDK's might help to some extent[43].

The case where it is assumed that the application systems themselves take care of the discovery of sub-resources by using HATEOAS, for example, could work on some cases. However, since a centralized structure for handling the services exists, it would be ideal that it could handle things like this. After all, that could be thought of as its primary job.

On the last two cases, "just knows" and "externalize it", the application system based authorization and the authorization of sub-resources versus what resource should be available through what Arrowhead application system, will cause one extra level of pain in cases where the sub-resources have different sets of authorized consumers. For example, some sensor reading might not be for all eyes, yet it might otherwise make sense to group it as a sub-resource with some other sub-resource, which is again for a different set of eyes.

Gatekeeper and Gateway

While it must be stated that the gatekeeper and the gateway systems gave the least amount of surprises compared to other systems they also had some problems, one from the more serious side is the incapability to establish a permanent session between the gateway systems at different local clouds, which forces an orchestration request, that goes through the whole process of inter-cloud orchestration, before each call. Ideally, the session would depend on the application systems lifetime.

It is also entirely possible that the incapability to achieve a permanent session is not due to the Arrowhead but rather due to the broker that was used between the gateway systems. Testing of this particular feature was left at the level of trying to get the gateways to understand "keep-alive" headers, without any success. Further configuration of the broker was not tried. However, if "non-stock" configuration of the broker is needed, it would be nice if it was documented somewhere.

One point worthy of mentioning on the gatekeeper system is its usage of HTTP for communication, which in practice means that the gatekeeper from the providing side must be visible to the gatekeeper at the consuming side. Although, since the gateway systems are using the broker, the only thing that needs to be visible in their communication is the broker. Could the gatekeeping also be moved to the broker [2]? This would enable more powerful tunnelling, which would have benefited the demo application by removing the need to poll.

[2] It turns out, that yes, indeed it can be moved. In version 4.1.3 that got finalised right before this thesis was finished the gatekeeping was moved to the broker [4].

If something like this was implemented, the way how other clouds are discovered would be needed to change. The current way, where the addresses, ports and gatekeeper URIs of local clouds at the neighbourhood are defined in the MySQL database would not be enough, instead if RabbitMQ [50] was used, the ids of the queues used by the broker would have to be stored instead, or otherwise discovered at both ends of the broker.

Application System Development

From the application system developments perspective, the most major shortcoming is the lack of libraries for interaction with the core systems. There are reference implementations of application systems, written both in Java and in C++, which can be used as a template for new systems, but this is far from ideal. Currently, the best — and if Java or C++ are not used, the only — option for registration and orchestration of services provided or consumed by application system is to write the HTTP requests directly "by hand" with the help of some generic HTTP-library.

This unavoidably leads to a situation where everyone developing application systems for Arrowhead Framework, is effectively writing their own "micro library", which most likely is not fully leveraging the functionalities that, the framework could offer, especially in the future when the framework hopefully provides more functionality. If "official" libraries for the most common languages were available, people could collaborate on those instead of wasting their time on writing parallel implementations, or, use the saved time on the development of their application systems.

It is also mention-worthy to state that all core systems provide an OpenAPI document, which can be fetched through an HTTP request. OpenAPI documents can be used for SDK generation, and generator implementations that target most commonly used languages exist [43]. However, the OpenAPI documents offered by the core systems also contain information about services that are not meant to be used by application systems but rather by other core systems or tooling build for management. This means that SDK generated solely for application systems' purposes will have extra code that it does not need and should not even be authorized to use.

6.2 Summary of Evaluation

Since the demo application achieved the features of the ideal system presented in section 4.1.1, a short answer to both research questions is that features offered by the Arrowhead Framework can indeed help on both the configuration of the data flow and the installation of new edge devices. Like it was assumed, especially the service discovery, which allows late binding and the tunnelling features offered by gatekeepers and gateways were both found useful, at least on the conceptual level.

However, the Arrowhead Framework can not be considered as a solution to the problems that the questions were derived from in section 1.2. This is mainly due to the infancy of

the platform, and there still is quite a lot of work to be done, if the Arrowhead Framework wants to be a contender in the field of automation and IoT.

Furthermore, container-based technologies can give the Arrowhead Framework pressure, since there is an urge to bring the container technologies used at the cloud also to the edge of the network. One example of a potential project is KubeEdge[32], which aims to extend the Kubernetes [33] platform to the edge of the network. These kinds of solutions, that do not only enable the service discovery via DNS but also provide means for their deployment in a sense of actually getting them up and running, which the current implementation of the Arrowhead Framework is not capable of doing, might be too big of a beast for the Arrowhead Framework to compete against.

The sections 6.1.2 above enlisted issues that were found during the development of the demo application, the point of view was especially on findings that would have an impact if the Arrowhead Framework was used in a production setup. However, many minor but still essential things like, the heavy memory usage, the bad quality of the code and general lack of robustness which, for example, manifests as a need to start the core systems in a "correct order" with external timeout scripts, to get the thing working in the first place, were not mentioned.

However, the framework's development process continues, and since it already has proven to have some upside and lot of the issues are fixable, an eye should be kept on the future versions of the framework. Since the development process already has continued for years, an ultra optimistic attitude towards the framework's future evolution might be too much to ask for, but in today's world, you never know.

7 CONCLUSION

In this master's thesis, an IoT framework known as the Arrowhead Framework was evaluated in a condition monitoring setup, by developing a demo application. The goal was to find whether the framework could help in 1) configuring the data flow from the edge to the cloud, and 2) the installation of new devices at the edge.

The demo setup consisted of a vibrating screen exciter and a set of vibration sensor boxes equipped with Bluetooth LE radios and accelerometers. The application was tested with an X86 industrial computer that served as the edge computer, and a virtual machine instance in Microsoft Azure cloud. Both, the edge computer and the VM had an Arrowhead local-cloud instance, with the mandatory Arrowhead core systems, the supporting core systems available with version 4.1.2 of the framework and a set of application systems that were implemented as Node.js applications.

A set of supporting tooling was needed. Including; PostgreSQL database, TimescaleDB extension for the database, PostgREST which was used for revealing the schema of databases as a REST service and Docker, which was used to ease the deployment of both the supporting tooling and the various systems related to the Arrowhead Framework, including the application systems.

To ease the application system development process, both, those in the demo application and those possibly developed in the future, a module-based scheme for building docker images was introduced. The scheme is based on a two-staged build process, which enables the reuse of modules that can utilize the Arrowhead core services by registering the services provided and discovering the services consumed, automatically in the case of the registration and through a simple interface in case of the discovery.

The methodological model used in the evaluation was the design science research process [44], which offers a framework for design science [28]. The evaluation consisted of the implementation of a demo application, which tried to achieve functionalities of an ideal system, which were derived from the research questions4.1.1. Afterwards, the demo applications success was evaluated, found shortcomings of the Arrowhead Framework were reported, and some suggestions for improvements were given as feedback for future development iterations.

The demo application achieved the functionality of the ideal system. However, the Arrowhead Framework was not found to be a production-ready platform. Shortcomings were found in all core systems of the framework. Therefore, it can be concluded that

version 4.1.2 of the Arrowhead Framework did not offer enough functionality for tackling the problems of data flow and installation of new edge devices satisfactorily.

In the summary section of the evaluation chapter6, it was suggested that the future developments of the framework should be followed since it was found to have some potential. However, it might be that the framework never gets to a level where it could be seriously considered as a production-ready solution. However, since, The Productive4.0 project continues, and the Arrowhead Framework is the primary tool, as future work, it is suggested that the framework's usage as a component of the test-bench-platform for further development of the condition monitoring system at Metso could be continued. For that purpose, the framework has more or less proven its suitability, especially if the crudest shortcomings are fixed.

Some interesting cases could include research at the edge of the network, where a collaborative edge, in which multiple edge computers with various "customer" roles could be used to investigate the further possibilities that SOA centred edge computing could offer. Another case could be Arrowhead compliant sensors. Since, REST is the lingua franca of the framework, investigation on REST compatible protocols for more constrained use cases like CoAP [51]with IPv6 over Low-Power Wireless Personal Area Networks (6LoWPAN) based solutions[39], where one possibility could be 6LoWPAN over Bluetooth LE [40], for which most of the equipment needed for basic level test-setups already exists.

Aside from the Arrowhead Framework, various solutions built around the container technology might have potential in an edge-cloud setup. Docker itself has cluster management features in the form of Docker swarm [14]. On top of this, the Kubernetes project also might offer new possibilities [33]. While this thesis was written, the Kubernetes project released an interesting, fully open-source solution called KubeEdge [32], which has the edge cloud as its primary target. A further study on the potential of the container-based cluster technologies expanded to the edge of the network could bring up interesting possibilities in the condition monitoring systems of Metso's equipment.

REFERENCES

[1] P. Aditya, I. E. Akkus, A. Beck, R. Chen, V. Hilt, I. Rimac, K. Satzke and M. Stein. Will Serverless Computing Revolutionize NFV? English. *Proceedings of the IEEE* 107.4 (2019), 667–678.

[2] R. Ahmad and S. Kamaruddin. An overview of time-based and condition-based maintenance in industrial application. *Computers and Industrial Engineering* 63.1 (2012), 135–149.

[3] S. V. Amari, L. McLaughlin and H. Pham. Cost-effective condition-based maintenance using markov decision processes. English. IEEE, 2006, 464–469. ISBN: 0149-144X.

[4] Arrowhead. *Projects github-repository*. URL: https://github.com/arrowhead-f (visited on 07/13/2019).

[5] Arrowhead. *Projects web-page*. URL: https://arrowhead.eu/ (visited on 07/12/2019).

[6] Axios. *Projects npm-page*. URL: https://www.npmjs.com/package/axios (visited on 08/01/2019).

[7] H. P. Bloch and F. K. Geitner. *Machinery failure analysis and troubleshooting*. English. Repr. Vol. 2.;2; Houston, TX: Gulf, 1986. ISBN: 0872018725;9780872018723;

[8] M. Burhan, R. A. Rehman, B.-S. Kim and B. Khan. IoT Elements, Layered Architectures and Security Issues: A Comprehensive Survey. *Sensors* 18 (Aug. 2018).

[9] M. Cerrada, R.-V. Sánchez, C. Li, F. Pacheco, D. Cabrera, J. Valente de Oliveira and R. E. Vásquez. A review on data-driven fault severity assessment in rolling bearings. English. *Mechanical Systems and Signal Processing* 99 (2018), 169–196.

[10] F. Civerchia, S. Bocchino, C. Salvadori, E. Rossi, L. Maggiani and M. Petracca. Industrial Internet of Things monitoring solution for advanced predictive maintenance applications. *Journal of Industrial Information Integration* 7 (2017). Enterprise modelling and system integration for smart manufacturing, 4–12. ISSN: 2452-414X.

[11] Control-Groups. *Manual-page(7)*. URL: http://man7.org/linux/man-pages/man7/namespaces.7.html (visited on 08/01/2019).

[12] CouchDB. *Projects web-page*. URL: http://couchdb.apache.org/ (visited on 09/20/2019).

[13] J. Delsing. *IoT automation : arrowhead framework*. Boca Raton: CRC Press, Taylor & Francis Group, 2017. ISBN: 978-1-4987-5675-4.

[14] Docker. *Projects web-page*. URL: https://www.docker.com/ (visited on 08/01/2019).

[15] S. O. Duffuaa, M. Ben-Daya, K. S. Al-Sultan and A. A. Andijani. A generic conceptual simulation model for maintenance systems. English. *Journal of Quality in Maintenance Engineering* 7.3 (2001), 207–219.

[16] ECMAScript. *Standards web-page*. URL: https://www.ecma-international.org/publications/standards/Ecma-262.htm (visited on 08/01/2019).

[17] M. W. Eder. Hypervisor-vs . Container-based Virtualization. 2016.

[18] Epoll. *Manual-page(7)*. URL: https://www.unix.com/man-page/linux/7/epoll/ (visited on 07/22/2019).

[19] T. Erl, B. Carlyle, C. Pautasso and R. Balasubramanian. *SOA with REST: Principles, Patterns & Constraints for Building Enterprise Solutions with REST*. English. 1st ed. Prentice Hall, 2012. ISBN: 0137012519;9780137012510;

[20] Express. *Projects web-page*. URL: https://expressjs.com/ (visited on 08/01/2019).

[21] Express-openapi. *Projects NPM-page*. URL: https://www.npmjs.com/package/express-openapi (visited on 09/22/2019).

[22] R. T. Fielding. REST: Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation. University of California, Irvine, 2000. URL: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

[23] R. Gao, L. Wang, R. Teti, D. Dornfeld, S. Kumara, M. Mori and M. Helu. Cloud-enabled prognosis for manufacturing. English. *CIRP Annals - Manufacturing Technology* 64.2 (2015), 749–772.

[24] C. M. GARCIA and R. ABÍLIO. Systems Integration Using Web Services, REST and SOAP: A Practical Report. English. *Sistemas de Informação* 1.19 (2017), 34–41.

[25] J. Gubbi, R. Buyya, S. Marusic and M. Palaniswami. Internet of Things (IoT): A vision, architectural elements, and future directions. English. *Future Generation Computer Systems* 29.7 (2013), 1645–1660.

[26] J. Halme, E. Jantunen, D. Hastbacka, C. Hegedus, P. Varga, M. Bjorkbom, H. Mesia, R. More, A. Jaatinen, L. Barna, P. Tuominen, H. Pettinen, M. Elo and M. Larranaga. Monitoring of Production Processes and the Condition of the Production Equipment through the Internet. English. IEEE, 2019, 1295–1300.

[27] A. Heng, S. Zhang, A. C. C. Tan and J. Mathew. Rotating machinery prognostics: State of the art, challenges and opportunities. English. *Mechanical Systems and Signal Processing* 23.3 (2009), 724–739.

[28] A. R. Hevner, S. T. March, J. Park and S. Ram. Design Science in Information Systems Research. English. *MIS Quarterly* 28.1 (2004), 75–105.

[29] A. K. S. Jardine, D. Lin and D. Banjevic. A review on machinery diagnostics and prognostics implementing condition-based maintenance. English. *Mechanical Systems and Signal Processing* 20.7 (2006), 1483–1510.

[30] Java. *Projects web-page*. URL: https://www.java.com/en/ (visited on 09/20/2019).

[31] Jie, W. Yu, N. Zhang, X. Yang, H. Zhang and W. Zhao. A Survey on Internet of Things: Architecture, Enabling Technologies, Security and Privacy, and Applications. English. *IEEE Internet of Things Journal* 4.5 (2017), 1125–1142.

[32] KubeEdge. *Projects web-page*. URL: https://kubeedge.io/en/ (visited on 09/28/2019).

[33] Kubernetes. *Projects web-page*. URL: https://kubernetes.io/ (visited on 09/28/2019).

[34] Linux-Namespaces. *Manual-page(7)*. URL: http://man7.org/linux/man-pages/man7/namespaces.7.html (visited on 08/01/2019).

[35] N. P. Manager. *Projects web-page*. URL: https://www.npmjs.com/ (visited on 08/01/2019).

[36] F. Mattern and C. Floerkemeier. From the Internet of Computers to the Internet of Things. English. Vol. 6462. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, 242–259. ISBN: 0302-9743.

[37] P. Mell and T. Grance. *The NIST Definition of Cloud Computing*. English. 2010.

[38] MongoDB. *Projects web-page*. URL: https://www.mongodb.com/ (visited on 09/20/2019).

[39] G. Montenegro, N. Kushalnagar, J. Hui and D. Culler. *Transmission of IPv6 Packets over IEEE 802.15.4 Networks*. RFC 4944. RFC Editor, Sept. 2007.

[40] J. Nieminen, T. Savolainen, M. Isomaki, B. Patil, Z. Shelby and C. Gomez. *IPv6 over BLUETOOTH(R) Low Energy*. RFC 7668. RFC Editor, Oct. 2015.

[41] Node-OPC. *Projects web-page*. URL: https://node-opcua.github.io/ (visited on 08/01/2019).

[42] Node.js. *Projects web-page*. URL: https://nodejs.org (visited on 07/22/2019).

[43] OpenAPI. *Projects web-page*. URL: https://swagger.io/specification/ (visited on 09/20/2019).

[44] K. Peffers, T. Tuunanen, C. Gengler, M. Rossi, W. Hui, V. Virtanen and J. Bragge. The design science research process: A model for producing and presenting information systems research. *Proceedings of First International Conference on Design Science Research in Information Systems and Technology DESRIST* (Feb. 2006).

[45] A. Pérez, G. Moltó, M. Caballer and A. Calatrava. Serverless computing for container-based architectures. English. *Future Generation Computer Systems* 83 (2018), 50–59.

[46] PostgreSQL. *Projects web-page*. URL: https://postgresql.org (visited on 07/14/2019).

[47] PostgREST. *Projects web-page*. URL: https://postgrest.org (visited on 07/14/2019).

[48] Productive4.0. *Projects web-page*. URL: https://productive40.eu/ (visited on 07/12/2019).

[49] Python. *Projects web-page*. URL: https://www.python.org/ (visited on 09/20/2019).

[50] RabbitMQ. *Projects web-page*. URL: https://www.rabbitmq.com/ (visited on 09/22/2019).

[51] Z. Shelby, K. Hartke and C. Bormann. *The Constrained Application Protocol (CoAP)*. RFC 7252. RFC Editor, June 2014.

[52] W. Shi, J. Cao, Q. Zhang, Y. Li and L. Xu. Edge Computing: Vision and Challenges. English. *IEEE Internet of Things Journal* 3.5 (2016), 637–646.

[53] Swagger-node-codegen. *Projects NPM-page*. URL: https://www.npmjs.com/package/swagger-node-codegen (visited on 09/22/2019).

[54] Systemd. *Projects web-page*. URL: https://www.freedesktop.org/wiki/Software/systemd/ (visited on 09/20/2019).

[55] TimescaleDB. *Projects web-page*. URL: https://timescale.com/ (visited on 07/14/2019).

[56] A. H. C. Tsang. Condition-based maintenance: tools and decision making. English. *Journal of Quality in Maintenance Engineering* 1.3 (1995), 3–17.

[57] P. Varga, F. Blomstedt, L. L. Ferreira, J. Eliasson, M. Johansson, J. Delsing and I. Martínez de Soria. Making system of systems interoperable – The core components of the arrowhead framework. English. *Journal of Network and Computer Applications* 81 (2017), 85–95.

[58] M. Weiser. The computer for the 21st Century. English. *IEEE Pervasive Computing* 1.1 (2002), 19–25.

[59] M. Wollschlaeger, T. Sauter and J. Jasperneite. The Future of Industrial Communication: Automation Networks in the Era of the Internet of Things and Industry 4.0. English. *IEEE Industrial Electronics Magazine* 11.1 (2017), 17–27.

[60] D. Wu, S. Liu, L. Zhang, J. Terpenny, R. X. Gao, T. Kurfess and J. A. Guzzo. A fog computing-based framework for process monitoring and prognosis in cybermanufacturing. *Journal of Manufacturing Systems* 43 (2017), 25–34. ISSN: 0278-6125.

[61] L. D. Xu, W. He and S. Li. Internet of Things in Industries: A Survey. *IEEE Transactions on Industrial Informatics* 10.4 (Nov. 2014), 2233–2243. ISSN: 1551-3203.

[62] L. D. Xu, E. L. Xu and L. Li. Industry 4.0: state of the art and future trends. English. *International Journal of Production Research* 56.8 (2018), 2941–2962.

[63] P. Zheng, H. Wang, Z. Sang, R. Y. Zhong, Y. Liu, C. Liu, K. Mubarok, S. Yu and X. Xu. Smart manufacturing systems for Industry 4.0: Conceptual framework, scenarios, and future perspectives. English. *Frontiers of Mechanical Engineering* 13.2 (2018), 137–150.